Universitat Politècnica de Catalunya (UPC) · BarcelonaTech

Multi-Agent Pathfinding for Unmanned Aerial Vehicles

Kymry Burwell

Thesis - Advanced Computing

Master in Innovation and Research in Informatics

Supervisor: Helmut Prendinger (Digital Content and Media Science Research Division, NII)

Tutor: Jordi Petit (Department of Computer Science, UPC)

BARCELONA SCHOOL OF INFORMATICS

The research and development work was carried out at the National Institute of Informatics (NII) in Tokyo, Japan.

Abstract

Unmanned aerial vehicles (UAVs), commonly known as drones, have become more and more prevalent in recent years. In particular, governmental organizations and companies around the world are starting to research how UAVs can be used to perform tasks such as package deliver, disaster investigation and surveillance of key assets such as pipelines, railroads and bridges. NASA is currently in the early stages of developing an air traffic control system specifically designed to manage UAV operations in low-altitude airspace. Companies such as Amazon and Rakuten are testing large-scale drone deliver services in the USA and Japan.

To perform these tasks, safe and conflict-free routes for concurrently operating UAVs must be found. This can be done using multi-agent pathfinding (mapf) algorithms, although the correct choice of algorithms is not clear. This is because many state of the art mapf algorithms have only been tested in 2D space in maps with many obstacles, while UAVs operate in 3D space in open maps with few obstacles. In addition, when an unexpected event occurs in the airspace and UAVs are forced to deviate from their original routes while inflight, new conflict-free routes must be found. Planning for these unexpected events is commonly known as contingency planning. With manned aircraft, contingency plans can be created in advance or on a case-by-case basis while inflight. The scale at which UAVs operate, combined with the fact that unexpected events may occur anywhere at any time make both advanced planning and planning on a case-by-case basis impossible. Thus, a new approach is needed. Online multi-agent pathfinding (online mapf) looks to be a promising solution. Online mapf utilizes traditional mapf algorithms to perform path planning in real-time. That is, new routes for UAVs are found while inflight.

The primary contribution of this thesis is to present one possible approach to UAV contingency planning using online multi-agent pathfinding algorithms, which can be used as a baseline for future research and development. It also provides an in-depth overview and analysis of offline mapf algorithms with the goal of determining which ones are likely to perform best when applied to UAVs. Finally, to further this same goal, a few different mapf algorithms are experimentally tested and analyzed.

Contents

Nomenclature

UAV related

UAV - unmanned aerial vehicle
Route - synonymous to path used in mapf nomenclature (path and route may be used interchangeably)

Pathfinding related

mapf - multi-agent pathfinding
sapf - single-agent pathfinding
CBS - conflict based search
GCBS - greedy conflict based search
ECBS - enhanced conflict based search
ICTS - increasing cost tree search
BFS - best first search
ID - independence detection
OD - operator decomposition
OID - online independence detection
SubID - Suboptimal online independence detection

1 Introduction

According to a 2016 study by NASA [Kopardekar et al., 2016] the number of unmanned aerial vehicle (UAV commonly known as drones), operations in the United States will grow to 2.7 million per year by 2020. An operation is defined as a single flight of a UAV. Another NASA study [Steve Bradford, 2018] estimates this number to be 3-7 million by 2021. The study also postulates the number of *daily* operations could potentially reach into the millions once legislation allowing package delivery is passed. These numbers include UAV operations carried out by hobbyists as well as private companies.

Generating the routes for all of these operations calls for automation using efficient multi-agent pathfinding (mapf) algorithms. A wide variety of such algorithms exist, but it is not clear which ones will perform best when applied to UAVs. The correct choice depends on many factors, including the structure of the map in which agents operate. Existing mapf algorithms have been experimentally tested on various maps in 2D space, but not in 3D, leaving the correct choice unclear. One of the goals of this thesis is to conduct a thorough review of state of the art mapf algorithms in order to help determine which may adapt well to the open 3D space in which UAVs operate.

When an unexpected event occurs in the airspace and a UAV is forced to change its route while inflight, new conflict-free routes must be found. This may be due to a medical helicopter occupying the airspace, police activity, or sudden high winds for example. Planning for these unexpected events is known as contingency planning in the aviation industry. With manned aircraft, contingency planning is typically done as a combination of advanced planning and with pilots on a case-by-base basis. The scale at which UAVs are projected to operate at make this type of contingency planning on a case-by-case basis impossible. Thus, a new approach is needed. Online mapf is one solution that looks promising. Online mapf utilizes traditional mapf algorithms to perform pathfinding in real-time. That is, new routes for UAVs can be found while inflight. There has been some recent research into online mapf, namely [Ma et al., 2017] and [Švancara et al., 2019], however, that research was conducted assuming agents are robots operating in 2D space.

With all of that said, there are three goals this thesis aims to accomplish:

(1) Provide an in-depth review of the properties, strengths and weakness of multi-agent pathfinding algorithms as well as an analysis of state of the art mapf and online mapf algorithms. The goal here is to help determine which algorithms, or types of algorithms, are theoretically best suited for use with UAVs, specifically taking into account that they operate in wide open 3D maps with few obstacles.

(2) Implement and experimentally test a few different mapf algorithms in 3D space. Many state of the art mapf algorithms have been experimentally tested, but only in 2D space. The goal is to help determine which ones experimentally perform well when applied to UAVs.

(3) Algorithms designed to perform inflight rerouting of UAVs will be necessary in the near future, however, there doesn't seem to be much research done in this field. A final goal of this project is to provide one possible implementation of a real-time inflight rerouting algorithm for UAVs that can serve as a baseline algorithm for future research. This implementation will also be experimentally tested in order to prove that a sophisticated approach is necessary.

The research and development work for this thesis was done during my internship at Professor Helmut Prendinger's laboratory at the National Institute of Informatics in Tokyo, Japan. Prof. Prendinger's laboratory is currently working under a Japanese government contract with the New Energy and Industrial Technology Development Organization (NEDO) to build prototypes of various UAV related systems. My job was to design and build a system for the inflight rerouting of UAVs. The laboratory team at NII, along with a previous intern, built out a basic version of a system used to perform offline mapf of UAVs. My work was a continuation and extension of this to real-time rerouting. The research and design of this real-time rerouting system, which utilizes two new frameworks and a few mapf algorithms was done independently by me. Further details of my contributions can be found in the conclusion, section 6.

This thesis is organized as follows:

Section 2 gives an in-depth overview of multi-agent pathfinding and provides a detailed summary of the algorithms implemented for this thesis. Section 2.7 introduces online multi-agent pathfinding and discusses the current state of the field. Section 3 presents the real-time inflight rerouting strategy developed for this thesis. Section 4 provides implementation details, including map, object and agent representation and design decisions made along the way. All development and experimentation was done in Java. Section 5 provides experimentation details and results. Section 6 is a conclusion with suggestions for future work.

2 State of the Art

2.1 Single-Agent Pathfinding

Before getting into the main topic of this thesis, a brief introduction to pathfinding is presented. Pathfinding is the process of searching for conflict-free paths between two points, typically with the goal of minimizing path length in terms of a cost function. Pathfinding is defined using a graph, starting at one vertex and ending at another, exploring adjacent nodes until the goal vertex is reached. During search, the path must avoid all obstacles and agents that may occupy vertices and edges of the graph.

There are many different search strategies that can be employed to find a path. Depth-first and breadth-first search find paths by exploring the entire graph and are therefore very slow for large graphs. Best-first search explores the graph using an evaluation function, $f(n)$, where n is the current node, to help guide the search by choosing to expand nodes that are most likely to lead to an optimal path quickly. Greedy best-first search uses a heuristic function, $h(n)$, to guide the search.

A* search is an iconic optimal pathfinding algorithm and it is worth our time to take a brief tour of it here as its still used for the low level (single-agent) search in many state of the art multi-agent pathfinding algorithms. It is possible to use it for multi-agent search but in reality is only used for single-agent search due to its exponential node expansion of b^d, where b is the branching factor, the number of successors generated by a given node, and d is the depth of the search tree. b is defined as b_{base}^k where b_{base} denotes the branching factor of a single agent and k is the number of agents. For example, in a two-dimensional search grid that allows diagonal movement with 3 agents, $b = b_{base}^k = 9^3 = 729$, so the number of nodes expanded is $O(729^d)$, a prohibitively large number even in small maps.

A*, as with many search algorithms, utilizes a cost function that includes a heuristic to guide the search by estimating the remaining cost to reach the goal, $f(n) = g(n) + h(n)$. $g(n)$ is the real cost of the path explored so far and $h(n)$ is the heuristic. A heuristic is admissible if it never overestimates the cost to reach the goal. The choice of heuristic is an important one, as heuristics that estimate poorly will force the search to expand more nodes, therefore increasing time and memory consumption. The diagonal distance heuristic is one of the more prominent heuristics and is the total distance from the agent's current position to the goal using either Manhattan or Euclidean distance, ignoring all obstacles. It typically performs quite well, however in an open search space with few obstacles it can become a perfect heuristic and cause the search to explore more paths

than necessary. A perfect heuristic is one that provides the actual minimum cost to the goal. In many cases, an agent will have many minimum cost paths to its goal and if a perfect heuristic is used, all of these paths will be explored before reaching the goal. To overcome this, a weight can be added to the heuristic function, forcing the search to favor nodes that are closer to the goal. The new cost function becomes $f(n) = g(n) + w \cdot h(n)$.

2.2 Multi-Agent Pathfinding Properties

The objective of a multi-agent pathfinding (mapf) problem is to find paths for multiple agents such that every agent reaches its goal and agents do not collide, typically with a goal of minimizing path length. Mapf can be applied to a variety of problem domains such as warehouse management, video games, map navigation, and aircraft flight paths.

Formal Definition

A mapf problem is be defined on an undirected graph $G = (V, E)$ and a set of k agents, $\langle a_1, a_2, \ldots, a_k \rangle$. Each agent a_i has a start location $s_i \in V$ and a goal $g_i \in V$. At each discrete time step an agent occupies a vertex $v \in V$ in the graph and can either move to an adjacent vertex or remain in the same vertex. Two agents cannot occupy the same vertex at the same time step nor can they cross the same edge e at the same time step. Each time step is a k-tuple $\langle p_0(t), p_1(t), \ldots, p_k(t) \rangle$ for which $p_i(t)$ is the position in the graph agent i occupies at time step t. A conflict between two agents is defined as a 4 tuple $\langle a_i, a_j, v, t \rangle$ in which agents i and j conflict at vertex v at time step t. That is, they both attempt to occupy the vertex during the same time step. This representation can be arbitrarily extended to a conflict between any number of agents.

Distributed vs Cooperative

Mapf problems can categorized into two general categories: distributed and cooperative. Also known as coupled and decoupled, respectively. In the distributed approach, each agent is responsible for finding its own path. Typically, a distributed approach occurs in three phases [Chouhan and Niyogi, 2015]: (1) agents individually find paths (2) a priority is given to the agents (3) the original plans are restructured according to the given priority. In step (3) the agent with the highest priority keeps its path, the agent with the second highest priority then finds a path with the added constraints of the first agent's path. This process continues for all agents in decreasing order of priority. Because

of this priority based method of deconfliction, distributed mapf problems are not complete and not guaranteed to find optimal solutions. A complete mapf algorithm is one that is guaranteed to find a solution if it exists. In the cooperative approach, a centralized planner computes the paths of all agents simultaneously with full knowledge of the agents and the search space. In contrast to distributed approaches, centralized approaches are complete and able to find optimal solutions.

Quality Measurements

The overall quality of a solution is generally based on two high-level cost functions: Sum of costs (SOC) and makespan [Surynek et al., 2016]. SOC is the summation of all individual agent path costs. That is $\sum_{i=1}^{k} C(a_i)$ where $C(a_i)$ is the cost of the full path for agent a_i. Makespan is the total number of time units required before the final agent reaches its goal. In other words, it is the cost of the longest path. All of the mapf solvers tested in this thesis use SOC.

Properties

When deciding which mapf algorithm is best for a specific problem, three important properties must be taken into account: completeness, optimality and soundness. A complete algorithm guarantees a solution will be found if one exists. In the case of pathfinding, completeness ensures all possible combinations of agents to positions for all time steps will be searched if necessary. Optimality ensures the algorithm terminates with an optimal solution with regards to a provided cost function. Soundness guarantees that a returned solution is correct. That is, all agent paths returned from a mapf solver are free of conflicts with both static obstacles and other agent paths. An algorithm having one of these properties does not imply anything about the other properties. For example, optimality does not imply soundness as an optimal algorithm may return false solutions along with an optimal solution, violating the soundness contract.

Solver Types

The aforementioned properties can be used to broadly define four types of mapf solvers: optimal solvers, suboptimal solvers, bounded suboptimal solvers and incomplete solvers.

Optimal solvers are sound, optimal and complete. Two prominent examples include Conflict Based Search [Sharon et al., 2015] and a solver that reduces the problem to SAT and uses existing SAT solvers for the final solution [Surynek et al., 2016].

Suboptimal solver are complete and sound but not optimal. Enhanced Conflict Based Search [Barer et al., 2014] and Cooperative A* [Silver, 2005] are two popular ones.

Bounded suboptimal solvers are complete, sound and provide a constant bound over the optimal solution cost. The algorithm takes a parameter w (weight) and return a solution that is less than or equal to $w \cdot C^*$ in which C^* is the cost of the optimal solution. The parameter w allows the trade-off between optimality and speed to be controlled by the user. ECBS and Weighted A* with are two examples.

Incomplete suboptimal solvers are not sound, complete nor optimal and are therefore not useful in pathfinding for UAVs. As such, they will not be discussed here.

Time Complexity

Ideally, optimal solvers would be used to solve every mapf problem. Unfortunately, optimality is computationally expensive in terms of both time and space. Solving mapf problems optimally has been proven to be NP-hard [Surynek et al., 2016]. Because of this, it is often necessary to use suboptimal solvers for problems with many agents. This leads to a trade-off between running time and the degree of suboptimality of the returned solutions. This trade-off must be evaluated on a case by case basis.

2.3 Multi-Agent Pathfinding Algorithm Categories

Multi-agent pathfinding algorithms can be categorized into three general categories, including rule based, reduction based, and search based solvers. Search based solvers are by far the most prominent, likely due to their ease of implementation and good solution quality to running time trade-off.

Rule Based

As the name implies, rule based solvers utilize specific rules based on the types of agents involved. These solvers typically find solutions quite fast, however the solution quality is often far from optimal [Felner et al., 2017] and many of the solvers only work on specific problem instances. TASS (tree-based agent swapping strategy) [Khorshid et al., 2011] for example is a sub-optimal complete algorithm that can solve problems in polynomial time, but with the limitation that it can only solve instances that can be converted to specific types of trees. The Push-and-Swap algorithm [Sajid et al., 2012] is proven to be complete only for trees, but is experimentally shown not to fail on general graphs provided there

is at least two unoccupied vertices. This algorithm works by allowing agents two basic operations: push and swap. Push enables the agent to move towards their goal and swap forces the agent to swap its position with another agent through the use of a free vertex. During this push and swap process the agents are given priorities in which an agent with higher priority is able to reserve a vertex for use in a swap action. These reserved vertices cannot be used by agents with lower priority. There is no bound on the optimality of the solution. BIBOX [Surynek, 2009] is complete only for bi-connected graphs — graphs in which any single vertex can be removed and the graph will remain connected. Again, there is no bound on the optimality of the solution.

Reduction Based

Reduction based solvers decompose the problem into other, usually well-known, problems such as SAT or integer linear programming (ILP). Commercial or open source solvers can then be used to solve the problem instances. Utilizing these solvers to do the bulk of the computational work is the main benefit to this approach, as solvers such as IBM's CPLEX for ILP problems, or MiniSAT for SAT problems have years of research and development put into them, making them efficient [Felner et al., 2017]. MDD-SAT, is a mapf algorithm that reduces the problem to a boolean satisfiability problem. The algorithm has been experimentally shown to be competitive with search-based solvers for hard problem instances, while search-based solvers prevail for easier instances [Surynek et al., 2016]. Another solver reduces the problem into a constraint satisfaction (CSP) problem [Ryan, 2010]. This problem experimentally works well on instances with a small number of agents (≤ 20 or so), but fails on larger instances.

Search Based

Search based solvers search a state space, typically represented as a graph, in a sequential manner while utilizing some type of search-based algorithm such as best-first search. A state is a mapping of all agents to locations (vertices in the graph) at a given time. Many search based solvers are based on the venerable A* (A-Star) algorithm [Hart et al., 1968] and each have their own strengths and weaknesses. Cooperative A* [Silver, 2005] for example is extremely fast even with many agents but is neither complete nor optimal. It works by assigning priorities to each agent and utilizes a reservation table of occupied states. The agent with the highest priority finds a path using simple A* and adds its path to the reservation table. The agent with the second highest priority then finds a path while avoiding the path of the first agent. This process continues for all agents. Enhanced

partial expansion A* (EPEA*) [Goldenberg et al., 2014] reduces the computational time and memory usage of A* by only generating child nodes whose $f()$ value is equal to the current nodes $f()$ value. That is, $f(n_c) = f(n)$ where n is the current node and n_c is a child of the current node. At each step, $f(n)$ is set to the $f()$ cost of the child node with the lowest $f()$ cost.

All of the aforementioned solvers can be seen as low-level solvers — algorithms that find the actual paths for the agents. Another type of search based solver utilizes a *high-level* and *low-level* algorithm during their search. High-level algorithms do not find the paths themselves, but rather guide the search in a promising direction while using the low-level algorithms to find the actual paths. Two of the best performing examples of these are conflict based search (CBS) [Sharon et al., 2015] and increasing cost tree search (ICTS) [Sharon et al., 2013].

2.4 Multi-Agent Pathfinding Algorithms

The following algorithms and frameworks will be discussed in detail because they were implemented and experimentally tested for this thesis: CBS, Greedy CBS, ECBS, A*, Focal A*, Operator Decomposition, Independence Detection.

Conflict Based Search (CBS)

Conflict based search (CBS) [Sharon et al., 2015] is an optimal, two-level, multi-agent pathfinding algorithm. At the high level, a search is performed on a tree based on conflicts between agents. At the low level, an optimal path is found using any optimal single-agent pathfinding algorithm. The conflicts produced in the high level are used as constraints in the low level. This two level decomposition reduces the state space of each individual agent search to be linear in the size of the graph. This ensures that each low level single agent search is very fast, however, there may be an exponential number of them produced by the high level. This is especially true if a group of agents is tightly coupled and produce many conflicts (more on this later). Even so, experimental results in [Sharon et al., 2015] have shown that CBS reduces the search time over traditional multi-agent pathfinding algorithms by up to an order of magnitude.

In CBS, a consistent path is defined as a path for a single agent that abides by all constraints imposed upon it by the high level. As previously defined, a conflict between two agents is defined as a 4 tuple $\langle a_i, a_j, v, t \rangle$ in which agents i and j conflict at vertex v at time step t. In CBS, only conflicts between two agents are considered - conflicts between

more than two agents are treated as multiple conflicts. For example, if agents i, j, k conflict at time step t on vertex v, three conflicts will be created: $\langle a_i, a_j, v, t\rangle$, $\langle a_i, a_k, v, t\rangle$, $\langle a_j, a_k, v, t\rangle$. A constraint is defined as a 3 tuple $\langle a_i, v, t\rangle$ in which agent i is not allowed to occupy vertex v at time step t.

High-level Search: At the high level, conflicts between agents are found and added as constraints to a data structure called a constraint tree (CT). A CT is a binary tree in which each node consists of three elements: (1) a set of constraints (2) a solution - a set of optimal, consistent paths for all k agents (3) the total cost of the solution (typically using the sum of costs heuristic). The root of the CT tree is a set of k optimal agent paths with no constraints. Once the root has been generated, high level search begins by finding the earliest conflict between a pair of agents, $\langle a_i, a_j, v, t\rangle$. Two child nodes are then created based on these conflicts with one constraint added to each: $\langle a_i, v, t\rangle$ and $\langle a_j, v, t\rangle$. A new single agent path adhering to the new constraint is found for each of these nodes and the total cost of the node is updated. Conflict detection is again performed for the node with the lowest cost. If no conflicts are detected, a solution has been found and the algorithm terminates. If a conflict is found, this process repeats itself and two new child nodes are created. When a new child node is created, it is added to an open list stored in a minimum priority queue, which guides the high level search to an optimal solution.

Low-Level Search: Low level search involves finding an optimal path for an agent given a set of constraints. Any optimal single-agent pathfinding algorithm will do, A* being a good choice. Optimality is guaranteed due to the fact that optimal best-first searches are run on both the high and low-level searches.

Algorithm 1 Conflict Based Search - High Level

```
1  procedure CBS(agents A, map M)
2      root ← new_node()
3      root.constraints ← ∅
4      for a_i ∈ A do
5          root.solution ← low_level_search(a_i, M)
6      root.cost = sum_of_costs(root.solution)
7      OPEN ← root
8      while OPEN ≠ empty do
9          N ← OPEN.poll()
10         C ← find_earliest_conflict(N)
11         if C is ∅ then
12             return N.solution
13         for a_i ∈ C do
14             n ← new_node()                          ▷ a new node is created
15             n.constraints ← N.constraints + (a_i, v, t)
16             n.solution ← N.solution
17             n.solution.update(a_i) ← low_level_search(a_i, M)
18             n.cost = sum_of_costs(n.solution)
19             OPEN ← n
```

Algorithm 1 lists the pseudocode for conflict based search. In lines 2-3, a root note for the high-level CT tree is created. Next, a sapf algorithm is used to find paths for all agents (lines 4-5). Using these paths, the cost of the root node is calculated based on sum of costs (line 6). The root is then added to the open list. Lines 8-19 perform the high-level search. At each iteration, the node with the lowest cost is polled form the open list (line 9). Next, the earliest pair of agent conflicts are found (line 10). If there are no conflicts, a solution has been found and the solution is returned (lines 11-12). Else, a new CT node is created for each of the two agents in the conflict (lines 13-19). One new constraint is added to each node (line 15) and a new path adhering to this constraint is found for the agent using sapf (line 17). Finally, the total cost of the nodes is updated and the node is added to the open list (lines 18-19).

Greedy Conflict Based Search (GCBS)

As previously mentioned, although CBS can provide a significant speed up over many mapf algorithms, the high-level search is still exponential in the number of conflicts, thus it does not scale well when there are many conflicts between agents. To get around this, sub-optimal variants, including greedy CBS (GCBS) and enhanced CBS (ECBS) were created [Barer et al., 2014]. GCBS is a suboptimal, complete variant that allows a more flexible search in the high and/or low level in which nodes that are more likely to provide

a valid solution are chosen first, even if the cost is higher than other nodes. At each high-level step in CBS, the node with the lowest overall cost is chosen for expansion. In GCBS, this search is guided by one or more conflict heuristics, h_c. A few different heuristics include:

- h_1 number of conflicts encountered in a specific CT node
- h_2 number of agents that conflict at a CT node
- h_3 number of pairs of agents that conflict
- h_4 vertex cover created from a graph in which the nodes are the agents and edges are conflicts between them
- h_5 alternating heuristic in which 2 or more of the previous heuristics are cycled through.

For the low-level search, a simple suboptimal single-agent pathfinding algorithm should not be used, as doing so may create many more high level conflicts. Instead, conflict heuristics should also be used at the low level. Each node in the high-level contains a set of constraints that the agent associated with the node must abide by when finding a path. Agent positions not contained in this list are not considered as constraints. In order to speed up the high-level search and create fewer conflicts, agent positions in the low-level should be considered, but not treated as hard constraints. That is, during the low-level search, agents should prefer paths that minimize future conflicts, even at the expense of optimality. For example, if an agent a_i occupies vertex v at time t, $\langle a_i, v, t\rangle$, and agent a_j is searching for a path, agent a_j should avoid vertex v at time t if it can, although it is not required to do so.

Enhanced Conflict Based Search (ECBS)

ECBS further extends GCBS by using a bounded focal search at both the high and low levels. A single heuristic (typically h_1) is used for ECBS.

Focal Search: In this approach, two sets of nodes are maintained. The first set is the typical open list from a best-first search. The second set is the focal set consisting of a subset of the nodes in open. In *focal_ search(f_1, f_2)*, f_1 and f_2 are arbitrary functions. f_1 is used to populate the focal list and f_2 guides the search. Focal is then populated as follows: $f_1(n) \leq w * f_{1_{min}}$, where w is the provided weight and $f_{1_{min}}$ is the minimum value of any node in the open list. f_2 is used to the choose the next node from the focal list to be expanded.

In ECBS, *focal_ search(f_{high}, h_c)* is used to guide the high-level search where $f_{high}(n)$ is the cost of the current CT node n and h_c is one of the conflict heuristics used in GCBS. *focal_ search(f_{low}, h_c)* is used to guide the low-level search where $f_{low}(n)$ is the standard A* cost function, $f_{low}(n) = g(n) + h(n)$, and h_c is one of the conflict heuristics used in GCBS. Combing the high-level and low-level searches, the sub optimality is bounded as follows: $C^* \leq w_{low} \cdot w_{high} \cdot C$ in which w_{low} is the weight used in the low-level focal search and w_{high} is the weight used in the high-level.

ECBS extends focal search to allow for even more flexibility and guides the search using both node cost and the conflict heuristics used in GCBS. Let $f_{min}(i)$ be a lower bound on the optimal consistent path for agent i. Then $f_{min}(n) = \sum_{i=1}^{k} f_{min}(i)$ is the lower bound on the optimal solution to the problem. This will be known as LB (lower bound). When a new CT node is generated in the high level, LB is updated to reflect the new lower bound. The high level then uses this LB parameter to populate the focal list. That is, $focal\_list = (n|n \in open, n.cost \leq LB * w_{high})$. As can be seen, this allows the high level search to be guided by both cost and conflict heuristics bounding the solution by $C \cdot w_h$.

Independence Detection (ID)

Independence detection is a generic framework that separates agents into discrete sets and finds paths for each set independently until conflicts between agents are found, at which time the conflicting agents are merged [Standley, 2010]. ID can be used on top of virtually any mapf algorithm. As previously discussed, many mapf algorithms have running times exponential on the number of agents. The thought is that by solving multiple subproblems with a small number of agents, this exponential running time can be avoided. ID is also essential in the dynamic in-flight UAV rerouting process, which will be discussed in section 2.7.

ID is quite simple in theory and works as follows. Each agent is initially assigned to their own group and a path for all agents is found independently of all other agents. Conflict detection is then performed. If there are no conflicts, a solution has been found and the algorithm terminates. If a conflict is found between two groups g_i and g_j (note: groups, not necessarily single agents). If these groups have conflicted before, they are merged into a new group, $g_i = g_i \cup g_j$, and paths for agents in this group are found using a mapf algorithm. If the groups have not conflicted before, new paths are found for group g_i with the paths of agents in g_j added as constraints. If a new set of conflict free paths with the same overall cost is found, the groups are not merged and conflict detection is resumed. If a new set of paths cannot be found, group g_j then attempts to find new paths

with the paths of agents in g_i added as constraints. If a set of conflict free paths with the same overall cost is found, the groups are not merged. If not, the groups are merged, $g_i = g_i \cup g_j$ and new paths are found for this new group. This process continues until paths have been found for all agents/groups.

It is possible that all agents eventually end up in the same group, thus ID does not provide a guarantee of speedup for mapf algorithms. In practice however, a significant speed up is seen in most cases. This has been shown in [Standley, 2010] and in my own experiments.

Algorithm 2 Independence Detection

```
1  procedure ID(agents A)
2      for all agents a ∈ A do
3          g_i ← a                                  ▷ assign agent to new group
4          π_{g_i} ← ψ^∅_{g_i}                      ▷ find optimal path for agent
5      while g_i and g_j conflict do
6          if g_i, g_j conflicted before then
7              g_i ← g_i ∪ g_j
8              π_{g_i} ← ψ^∅_{g_i}
9          else if ψ^{π_{g_j}}_{g_i} as good as π_{g_i} then
10             π_{g_i} ← ψ^{π_{g_j}}_{g_i}
11         else if ψ^{π_{g_i}}_{g_j} as good as π_{g_j} then
12             π_{g_j} ← ψ^{π_{g_i}}_{g_j}
13         else
14             g_i ← g_i ∪ g_j
15             π_{g_i} ← ψ^∅_{g_i}
```

Algorithm 2 shows the general outline of ID. Some helpful notation is introduced. π_{g_i} is a set of conflict free paths for a group of agents g_i. $\psi_{g_j}^{\pi_{g_i}}$ is an optimal plan for a group of agents g_j while avoiding the plan for agents in g_i. It is assumed that the groups are disjoint. $\psi_{g_j}^{\emptyset}$ is an optimal plan for g_j while avoiding no other agent plans.

In lines 2-4, each agent is assigned its own group and optimal paths for all agents are found using sapf. Lines 5-15 iteratively finds and resolves conflicts between groups. When a conflict is found and the groups have conflicted before, the groups are merged and optimal paths are found together (lines 6-8). If the groups have not previously conflicted, group g_i searches for a set of optimal paths while avoiding avoiding the plans for agents in group g_j that does not increase the sum of costs (lines 9-10). If this cannot be done, g_j does the same while avoiding g_i (lines 11-12). If this cannot be done, the groups are merged and paths for the merged group are found together (lines 13-15).

Suboptimal Independence Detection (SubID)

Suboptimal independence detection (SubID) is simply a suboptimal version of ID that accepts solutions within a given factor D of the optimal solution. When resolving conflicts between groups (lines 9-11 in algorithm 2), plans with cost up to $D \cdot SOC(\psi_{g_j}^{\pi_{g_i}})$ are accepted.

Operator Decomposition (OD)

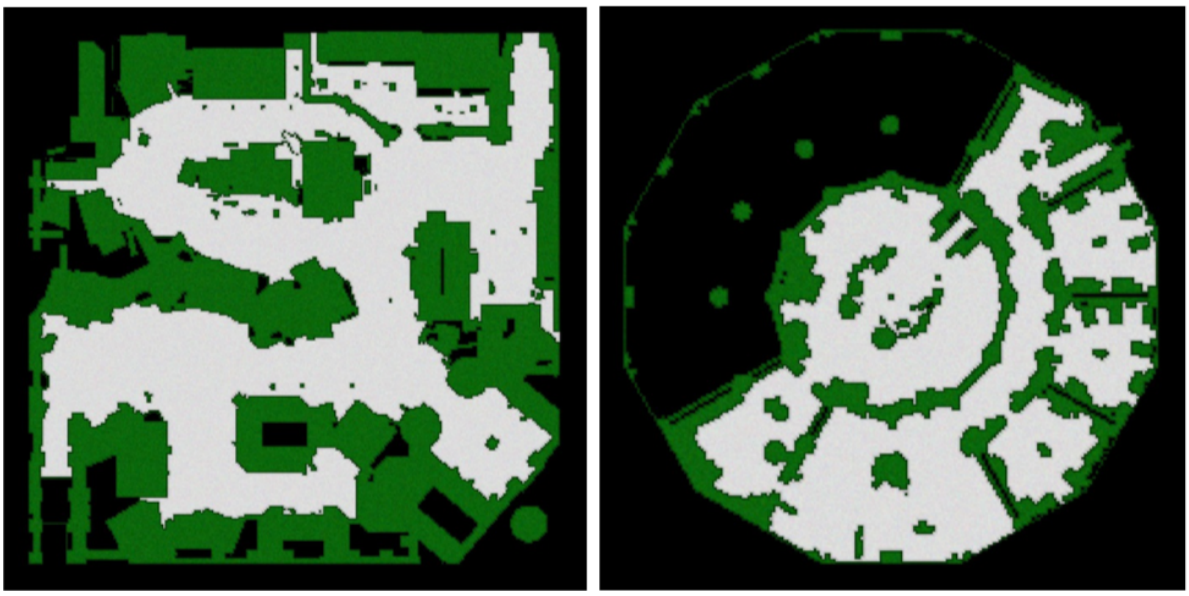
Operator decomposition (OD) is not an algorithm in itself, but rather an enhancement that can be added to A* based search algorithms. In most typical A* algorithms, all agents perform a single move at each step of the algorithm, creating an exponential branching factor of b^k. This is clearly much too slow for a large number of agents with even a moderate branching factor. OD attempts to alleviate this exponential node expansion by allowing only a single agent to make a move at each step of the algorithm. To do this, each time step of the algorithm is divided into k intermediate steps, one for each agent. This reduces the branching factor from b^k to just b. However, the depth of a goal node in the search tree increases by a factor of k. In standard A*, there is only a single type of node, the standard node. If OD is used, there are two types of nodes, standard nodes and intermediate nodes. In standard A*, standard nodes correspond to time steps in which all k agents make a move simultaneously. With OD, a standard node is a state in which no agent has made a move. An intermediate node is all the nodes in between standard nodes. At each intermediate node, a single agent is assigned a move. There are k intermediate nodes in between each standard node. To determine which agent makes a move first, a priority is given to the agents when the algorithm is instantiated.

A* with Operator Decomposition and Independence Detection (A*+OD+ID)

A* can be used with either operator decomposition or independence detection, or both. A*+OD is exactly as described in the OD section above, in which agents make their moves one at at time. A*+OD+ID combines this single agent move action with independence detection. At the start, every agent is assigned to their own group. Starting with the first agent, an optimal path is found for every agent. Once all agent paths have been found, conflicts are detected and conflicting agents are grouped as necessary. Agents that are grouped use A*+OD to find new paths. Experimental analysis in [Standley, 2010] has shown that A* is most performant in terms of running time and hardness of problems when used with both OD and ID.

2.5 Benchmark Testing Data

The benchmark data used for testing most of the above mentioned mapf algorithms come from the maps in a computer game named *Dragon Age: Origins* and was originally proposed in [Sturtevant, 2012]. Three specific maps from this game were used and are displayed in Figures 1 and 2. As can be seen, each of the maps has a varying density of obstacles, but each is far denser than the maps in which UAVs will operate. This makes theses maps unsuitable for testing mapf algorithms applied to UAVs. It also shows that many state of the art algorithms haven't been thoroughly tested in maps of varying structure, at least not in the original papers. This is one of the reasons the correct choice of mapf algorithms for UAVs remains unclear.



(a) DAO map: Den520 **(b)** Dao map: ost003d

Figure 1

Figure 2: DAO map: brc202d

2.6 Choosing an Algorithm

Choosing a suitable mapf algorithm for the problem at hand is not always (in fact, almost never) straightforward. Many factors can influence this choice: density of agents, number of agents, map structure (defined as the density and layout of obstacles within a map), optimality requirements and speed requirements. There does not seem to be a definitive consensus on which algorithm to choose under different configurations of theses factors, creating an excellent area of future research. This lack of consensus also led to one of the questions being addressed in this thesis — "What is a good mapf algorithm for use with UAVs in wide open maps with few obstacle and a varying number of overlapping paths?". One important factor that needs to be considered when answering this question is the differences between pathfinding in 2D and 3D space. On the one hand, hard mapf instances in 2D space will likely be easier to solve in 3D space, thanks to the extra dimension agents can occupy. On the other hand, this third dimension adds complexity to development and introduces more chances for bugs. With that said, finding conflict-free paths for UAVs is very important due to the dangers posed to humans with UAVs operating overhead. So even though many mapf problem instance will be solved easier in 3D space, the worst case (i.e. many densely operating agents) must be taken into account.

A couple of examples in which different configurations impact algorithm choice follow.

[Sharon et al., 2015] experimentally showed that CBS scales well with an increasing number of agents when compared to A*+OD and ICTS. He also showed that CBS works

well with most maps and particularly well on maps with bottlenecks and narrow corridors. This is due to CBS expanding fewer high level CT nodes and thus reducing the search time. When CBS expands a CT node, single agent paths are found for both agents, with constraints imposed for the conflicting agent. When a narrow corridor is reached and two agents need to pass through, CBS only needs to try two paths (one for each agent) before realizing that they cannot both path and keep the same path cost. At this point, CBS will increase the optimal cost of a path, allowing one of the agents to pass with this increased cost. [Sharon et al., 2013] experimentally shows that ICTS works well on maps with wide corridors that many agents can pass through simultaneously. [Felner et al., 2017] showed the MDD-SAT scales well with an increasing number of agents compared with ICTS and EPEA* in open maps. [Barer et al., 2014] showed that ECBS performed better than other A* variants EPEA* and RM* in nearly all map types with 20% agent density. [Surynek et al., 2016] compared ICTS, ECBS and MDD-SAT. Experiments showed MDD-SAT worked best with 16 or fewer agents on various maps, while ICTS performed best on the same maps with more agents.

These are just a few examples with a few algorithms, but it is clear that there is no consensus on which algorithm is best under specific conditions. However, there does seem to be general consensus in the mapf community that ICTS and ECBS are the two best general solvers at this time. Other solvers such as MDD-SAT excel under specific conditions, but not generally. This was one of the deciding factors when choosing CBS and its variant ECBS as the primary algorithm for experimentation.

2.7 Online Pathfinding

Most mapf algorithms are designed to work with a static set of agents in which the start and goal locations of each agent are predefined and unchanging. This works well in many cases, but it is sometimes necessary to add new agents after paths for the original agents have already been found or to dynamically change a goal location or path without prior knowledge of the impending change. These are very real challenges currently being tackled in both industry and academia. For example, in order to reduce human error and lower operating expenses, companies that operate warehouses, such as Amazon, are employing teams of automated robots to retrieve products within the warehouse to be shipped to customers. This is just one example of a multi-agent pathfinding problem in which new agents are continuously being added. For example, when a customer places an order, a new robot is deployed to retrieve the item from a shelf in the warehouse. A robot in this scenario may also change its goal location, for example when a new order is placed and

a currently operating robot is in the vicinity of the product that needs to be retrieved. Instead of deploying a new robot, an existing robot is simply rerouted.

As another example applied to UAVs, consider a situation where many UAVs are delivering packages and an emergency occurs in which a medical helicopter has to occupy the same airspace. In this case, the helicopter can be seen as either a new agent with priority or as an obstacle. In either case, some UAVs will likely need to change their routes and/or goal locations. This is depicted in figure 5. Figure 3 is a simplified example of an unexpected no-fly zone being added to the airspace, forcing a UAV to reroute. If the UAV reroutes using a sapf algorithm and doesn't take into account other agents, conflicts may occur. Figure 4 is a pictorial example of a situation in which an unexpected no-fly zone covers a UAVs goal location, showing the need for online mapf. This picture also introduces one idea surrounding UAVs not yet discussed, the *drone port*. A drone port is simply a specified location in which a UAV is allowed to enter and exit the airspace. A drone port is nearly identical to an airport, in which planes take off sequentially, not simultaneously.

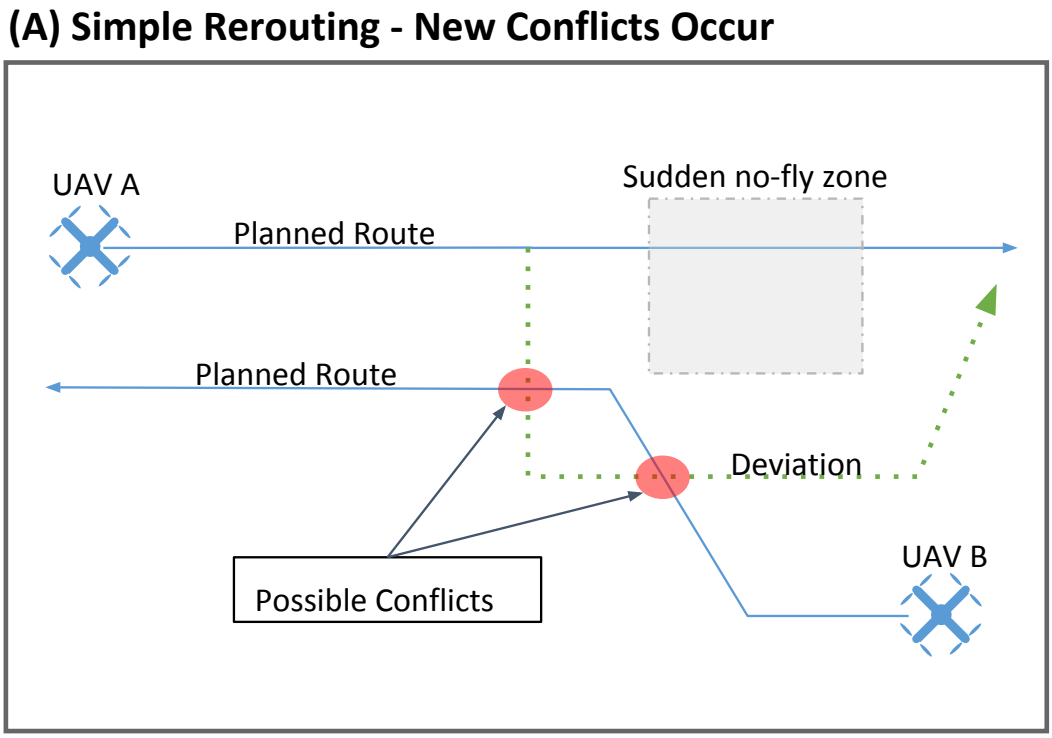


Figure 3: single-agent rerouting

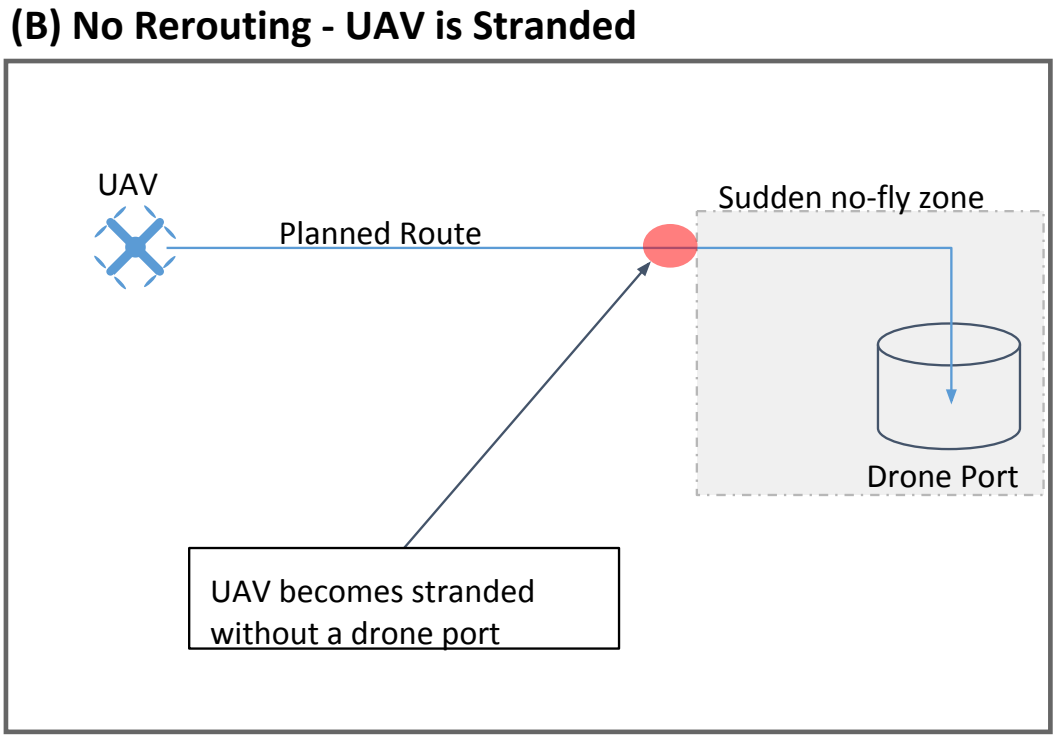


Figure 4: no rerouting

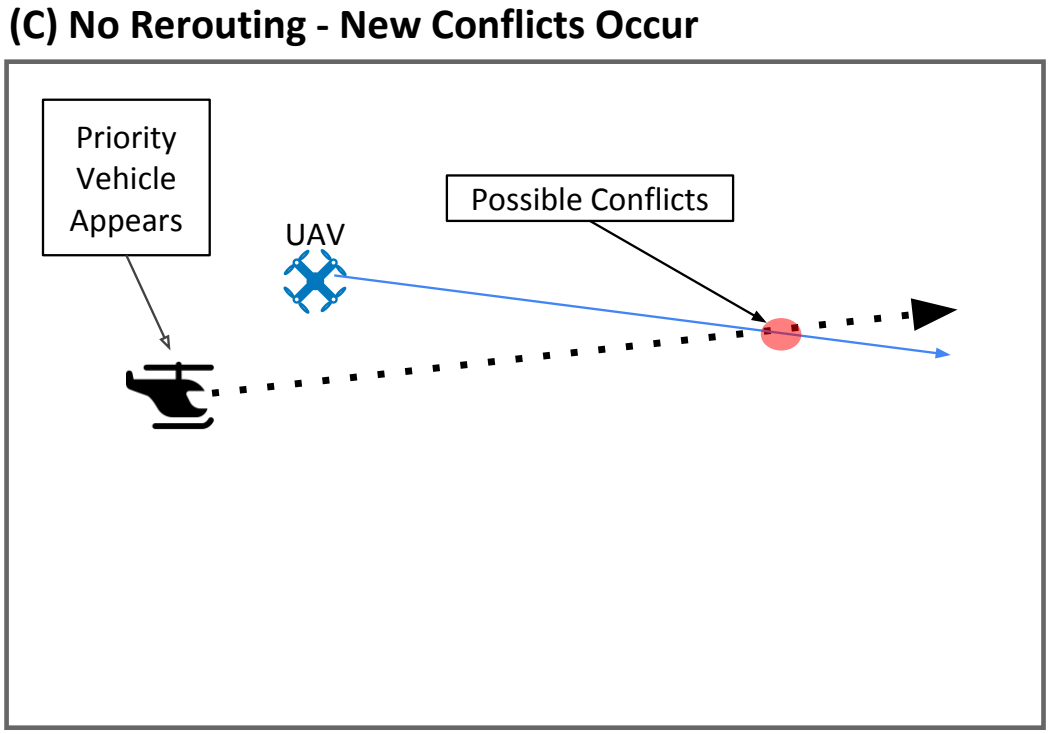


Figure 5: no rerouting

These types of scenarios in which agents are both added and change their paths and goal

locations dynamically without prior knowledge will be known collectively as online mapf. There has been some recent research done in this field, namely [Ma et al., 2017] and [Švancara et al., 2019].

Lifelong mapf

In [Ma et al., 2017], Ma and his team introduce two novel algorithms for solving the problem in which agents perform and exchange tasks in a confined area, which Ma refers to as *Lifelong mapf for online pickup and delivery tasks*. In this scenario, the set of agents is static — agents are neither added nor removed. Agents continually add new paths and change their existing paths throughout the day as new tasks arrive and as tasks are exchanged between agents. An example of this is automated vehicles at a shipping port removing shipping containers and moving them around a confined area. There is always the same set of vehicles but they may add new paths and exchange tasks as necessary.

Ma presents two algorithms, Token Passing (TP) and Token Passing with Task Swaps (TPTS). These are designed for problems in which there is a set of unexecuted tasks, $\mathcal{T}$, and a set of agents that perform these tasks. New tasks are added throughout the day. A task, $\tau_i \in \mathcal{T}$ is characterized as a pickup location $s_j \in V$ and a drop-off location $g_j \in V$. When a new task arrives it is assigned to an agent. The agent executes the task by going to the start location, picking up the object, going to the goal location, dropping off the object, and finally going to an unoccupied resting location. A resting location $r_k \in V$ is defined as a location where an agent remains until it is assigned a task. There are at least as many resting locations as there are agents. Additionally, it is always possible to get from one resting location to another without passing through a third resting location.

This is simply a special case of an mapf algorithm in which tasks (paths) are assigned continuously and therefore agents are constantly changing their locations dynamically. TPTS takes this one step further by allowing agents to swap tasks they are assigned but have not yet begun to execute. This is beneficial when another agent can perform a task more quickly than the agent currently assigned to the task. The details of the algorithms will not be covered here, but a good explanation can be found in Ma's paper.

Online mapf

In [Švancara et al., 2019], Svancara provides two contributions: (1) A formal definition and analysis of the online mapf problem and (2) two practical solvers for online mapf.

The solution for dynamic inflight UAV rerouting presented in this thesis was heavily influenced by this paper, so I shall take some time to discuss it in detail here. In offline mapf, the set of agents is unchanging and the paths for all agents are found before any agent begins execution of its path. In online mapf, new agents may appear after agents have begun to operate. A simple solution to this is to force the new agent to avoid all currently operating agents, perhaps through the use of a reservation table with A*. This method has two major flaws: (1) the new agent may never be able to reach its goal due to the constraints imposed by other agents and (2) the new agent's path may be far from optimal. Online mapf addresses these deficiencies and is defined in the following section.

Formal Definition

Online mapf is a natural generalization of mapf in which new agents are allowed to dynamically enter the mapf problem during runtime. In Online mapf, a conflict free path for an agent a_i is defined as π_i. At time step t, agent a_i is at position $\pi_i[t]$.

A solution to an online mapf problem is defined as $\Pi = \langle \pi^0, \pi^1, \ldots, \pi^m \rangle$, where m is the number of times a new agent appears. It is assumed that an agent appears at an unoccupied position when it appears. π^i is the set of paths created when the i^{th} agent appears. π^0 is the original set of paths created from the offline mapf solver.

A partial plan $\pi^i[t_x : t_y]$ is the part of π^i that is actually executed. That is, the paths defined in π^i are followed for time steps $x, x+1, \ldots, y$ only.

$Ex[\Pi] = \pi^0[t_0 : t_1] \circ \pi^1[t_1 : t_2] \circ \ldots \circ \pi^m[t_{m-1} : t_m]$ are the set of final paths all agents follow. Here, the $\circ$ symbol denotes concatenation. That is, agents follow path π^0 for time steps t_0 to t_1, then path π^1 for t_1 to t_2, etc.

Basically, each π^i is a set of full conflict free paths for each agent, but these paths are only followed until the next agent appears and new paths are created. Generating new paths for all agents whenever a new agent appears ensures snapshot optimality, which is defined in the next section.

Evaluation Metrics

In order to evaluate the solution quality of an online mapf problem, the Sum of Cost (SOC) measure will be used. SOC for online mapf is defined as the sum of costs for the set of executed plans $Ex[\Pi]$:

$$SOC(\Pi) = \sum_{i=0}^{m} Ex[\Pi]_i \tag{1}$$

where $Ex[\Pi]_i$ is the cost of the executed paths for the agents active when agent i appeared.

It should be noted that no complete online mapf solver can guarantee a solution with a cost equal to that of an offline solver. The proof of this can be found in [Švancara et al., 2019]. Even still, an online mapf solver can return a solution that is *snapshot optimal*. A snapshot optimal plan is one that is optimal in terms of SOC assuming no new agent appears. That is, in a solution to online mapf problem, $\Pi = \langle \pi^0, \pi^1, \ldots, \pi^m \rangle$, each set of paths π^i is optimal at the time it is created.

Algorithms Defined

RS, RSG, RA: There are a few different strategies that can be used to solve online mapf problems. Three such strategies are *Replan Single* (RS), *Replan Grouped* (RSG) and *Replan All* (RA). RS finds paths for all new agents, one at a time, while avoiding all agents whose paths have already been found. RSG finds paths for all new agents simultaneously, using any mapf algorithm, while avoiding all agents whose paths have already been found. RA replans for all agents simultaneously, already planned agents included. Since RS and RSG do not allow the replanning of already planned agents, the solution quality may be quite poor, if a solution can even be found. RA takes this to the other extreme, providing snapshot optimal solutions, but increasing running time significantly.

Online Independence Detection: As mentioned, RA plans optimally for all agents, completely ignoring previous paths. This can be wasteful in terms of memory and running time and may introduce unnecessary replanning. Suppose the map in which agents operate is large, and the paths of agents operating on one side of the map rarely, if ever, interact with agents on the other. If a new agent appears on one side, it is likely pointless to replan the agents on the other side. To overcome this deficiency, *independence detection* (ID) can be used. This will be called *online independence detection* (OID). In OID, paths for new agents can planned independently of all other agents. If there is a conflict with another agent during the planning process, the two (or more) agents are merged into a group and planned together. This process continues until conflict free paths for all agents are found. Groups persist throughout the life of an agent. The necessity of this can be seen with a simple example. Suppose a new agent a_3 conflicts with an existing agent a_1. It is not sufficient to simply group the two agents and replan for them together. It may

be the case that a_1 has already conflicted with another agent a_2 and a_2 took a longer path in order to avoid a_1. When a_1 is replanning, it may free up spaces in the state space that a_2 can utilize for a shorter path. As can be seen, if an agent has to replan multiple times and groups do not persist, it is possible that a path for an agent may get longer and longer. To avoid this situation, agent groups persist. Algorithm 3 below is almost exactly the same as ID presented in section 2 and therefore details will not be provided again here. The only difference being lines 2-6, in which existing agent groups are also included as input. These groups simply bypass lines 3-6. The output is a set of snapshot optimal conflict-free paths for all agents.

It is worth emphasizing that OID needs to be used in conjunction with mapf and sapf algorithms. This means that any limitations of the algorithms used are still present when using OID. For example, when a new agent is added and a path is being found, it may be that it wants to occupy the same goal location at the same time as another agent. This situation needs to be addressed in the pathfinding algorithm, not in OID. OID simply acts as a sort of framework that attempts to minimize the number of agents that need to be rerouted. Additionally, OID does not take into consideration any idea of fairness, meaning all agents are treated equally. However, it is possible to use algorithms that do consider fairness, such as Cooperative A*, which gives priorities to agents [Silver, 2005].

Algorithm 3 Online Independence Detection

```
1  procedure OID(agents A = ∪_{i=1}^{m} g_i, new agents A^+)
2      m ← m + 1
3      for a ∈ A^+ do
4          g_m ← a                                   ▷ assign agent to new group
5          π_{g_m} ← ψ^∅_{g_m}                       ▷ find path for agent
6          m ← m + 1
7      while g_i and g_j conflict do
8          if g_i, g_j conflicted before then
9              g_i ← g_i ∪ g_j
10             π_{g_i} ← ψ^∅_{g_i}
11         else if ψ^{π_{g_j}}_{g_i} as good as π_{g_i} then
12             π_{g_i} ← ψ^{π_{g_j}}_{g_i}
13         else if ψ^{π_{g_i}}_{g_j} as good as π_{g_j} then
14             π_{g_j} ← ψ^{π_{g_i}}_{g_j}
15         else
16             g_i ← g_i ∪ g_j
17             π_{g_i} ← ψ^∅_{g_i}
```

Suboptimal Online Independence Detection: If OID alone is still too slow, suboptimal OID (SubOID) may be used. SubOID is a relaxation of OID that returns

plans whose sum of costs is at most D times the optimal plan. When OID replans for an agent group, g_i while avoiding g_j, only plans with the same SOC as the optimal plan for group g_i are accepted. The optimal plan being $SOC(\psi_{g_i}^{\pi_{gj}})$. SubID allows plans with cost up to $D \cdot SOC(\psi_{g_i}^{\pi_{gj}})$ to be accepted. Recall from section 2.4 that $\psi_{g_i}^{\pi_{gj}}$ is an optimal plan for a group of agents g_i while avoiding the plan for agents g_j for a single invocation of independence detection. Throughout the life of an agent, independence detection may be called several times, with a new optimal (or suboptimal) path found each time.

Recap

To summarise, *lifelong mapf* is an extension of mapf in which new tasks, defined as agent paths with a start and end location, are dynamically added to an ongoing mapf problem. *Online mapf* is an extension of mapf in which new agents are dynamically added.

3 Dynamic Inflight UAV Rerouting

This section introduces the dynamic real-time rerouting problem applied to UAVs, provides a generic formalization, and presents the solution created for this thesis. The word route, as opposed to path, is used because it is the term most widely used by the UAV community,

Dynamic inflight UAV rerouting is a special case of online mapf applied to UAVs. In the UAV ecosystem, unexpected events may occur in the airspace, forcing autonomous UAVs to deviate from their designated routes. A UAV may experience engine failure and need to safely maneuver to a nearby landing location. A medical helicopter may need to occupy the same route as a UAV. Inclement weather or police activity may cause a large portion of the airspace to be temporarily closed. These are all events that dynamically add an obstacle to the mapf problem at runtime. The terms obstacle and no-fly zone will be used interchangeably. A UAV experiencing engine failure can be seen as an obstacle because it will not be forced to change its path and therefore must be avoided by other UAVs.

Every UAV has an operation. An operation consists of a start location, s_i, service location, l_i, and goal location, g_i. The start and goal locations can be seen as drone ports where UAVs take-off and land. They may or may not be the same location. The service location is where the UAV performs its task. A task is generic and can be anything. For example, a UAV delivering a package departs from s_i with the package, delivers the package to l_i and lands at g_i. Instead of delivering a package, the UAV may be tasked with taking pictures at the service location. No matter the task, the definition of an operation remains the same.

3.1 Challenges

When a dynamic no-fly zone is introduced into the airspace, all UAVs with routes intersecting the no-fly zone must be rerouted. Developing an effective strategy to perform this rerouting introduces a number of challenges.

- The algorithm must be fast, as agents are inflight.
- A path must always be found, even if not optimal or conflict free.
- Alternate paths cannot be predefined as dynamic no-fly zones can appear anywhere in the airspace.
- The algorithm must accommodate scenarios in which the agent goal location is in conflict with the no-fly zone.

- A sufficient number of alternative landing locations must be available to accommodate all agents. Imagine a large no-fly zone is introduced that prevents many agents from reaching their goal location and in turn new goal locations must be assigned.
- The the algorithm must be able to handle cases where agents are inside the vicinity of the dynamic no-fly zone when it is announced.

There are countless strategies that can be followed to overcome these challenges. The next section provides a formal definition and presents the strategy developed for this thesis.

3.2 Formal Definition

In order to formally define the rerouting problem, the characterization of online mapf from section 2.7 can be used with just a couple of slight modifications.

Recall that a solution to an online mapf problem is defined as $\Pi = \langle \pi^0, \pi^1, \ldots, \pi^m \rangle$, where m is the number of times a new agent(s) is added and π^i is a set of conflict free paths for all active agents at that time.

For the inflight rerouting problem, each $0, 1, ..., m$ is defined as any event that causes at least one agent path to be planned or replanned during runtime. This can occur when (1) a new agent(s) is added or (2) a dynamic obstacle(s) is added. Both of these events may occur simultaneously. The definition π^i remains the same, it is the set of conflict free paths found for all agents when one of these events occurs.

As in lifelong mapf, an agent's goal location may be changed during runtime. Therefore, a set of goal locations, $r^k \in V$ must be defined. Unlike lifelong mapf, there doesn't need to be as many goal location as agents. This is because in the rerouting problem agents disappear when they reach their goal location. This is a reasonable assumption when considering that UAVs are removed from the drone port after landing.

It should be noted that changing an agent's goal location doesn't affect the snapshot optimal solution quality metric as discussed in section 2.7.

3.3 Solution Defined

The solution presented below was developed as part of the work done for this thesis. Before jumping straight into the solution, let's look at a few different strategies to consider.

1. Use single-agent pathfinding to find new paths for all agents in conflict with the no-fly zone. The paths of agents not in conflict with the no-fly zone are not considered as constraints. This approach is very fast and ensures no agents conflict with the no-fly zone. However, optimal and conflict free paths between agents is not guaranteed as new paths may conflict with existing agents.

2. Use multi-agent pathfinding to find paths for all agents simultaneously. This option is the slowest to find a path for all agents, but ensures conflict free and snapshot optimal paths for all UAVs if they exist.

3. Combine the first two strategies. First, quickly reroute all agents in conflict with the no-fly zone(s) using sapf. Second, use mapf to solve conflicts between agents. If time runs out during execution of mapf, at least *some* path exists for every agent. This helps ensure that even if a no-fly zone is introduced very quickly, paths for all agents around the no-fly zone can likely be found in time. On a side note, if this happens, a collision detection and avoidance algorithm such as GRCA [Bareiss and van den Berg, 2015] can be used to overcome agent conflicts during flight. Collision detection and avoidance algorithms differ from pathfinding algorithms in that they do not plan paths, they simply take short evasive maneuvers in real-time to avoid a collision, only considering agents and obstacles in the close vicinity (e.g. 100 meters for a UAV). After the collision has been avoided, the UAV returns to the original path.

The solution developed for this thesis uses option 3. It is based on a combination of lifelong mapf and online mapf. Next, a few assumptions are made.

- There are enough landing locations to accommodate all UAVs. The number of landing locations needed will depend on how quickly a UAV is removed from the drone port once it lands and is problem dependent.

- Dynamic no-fly zones will be announced with sufficient time for UAVs to reroute. The amount of time necessary is problem dependent. This is a necessary requirement due to the fact that no-fly zones can appear anywhere and there may be agents inside the area where one is announced. These agents must be able to leave the no-fly zone as they cannot simply land anywhere.

- All UAVs are holonomic, meaning they can change direction instantaneously and hover. Figure 6 shows the two types of of drones: holonomic and non-holonomic (fixed wing).

(a) Holonomic Drone (Rakuten)[1]

(b) Fixed wing drone (Hussar)[2]

Figure 6

With these assumptions in mind, the dynamic inflight rerouting algorithm, which I'm calling *UAVReroute* (UAV Rerouting) is outlined in Algorithm 4.

Algorithm 4 UAVReroute — Dynamic Inflight Rerouting

```
1  procedure REROUTE(agents A, new agents A+, new obstacles O, map, rerouting
   time step t)
2      map.obstacles ← O                          ▷ add obstacles to map
3      find_conflicts(A, map)
4      for a ∈ A do
5          if a has goal location conflict then
6              assign_goal_location(a)
7              sapf(a, t)
8          else if a has service location or path conflict then
9              sapf(a, t)
10     for a ∈ A+ do
11         sapf(a, t)
12     try
13         OID(A, A+)                    ▷ Online Independence Detection
14     catch TimeOutException     ▷ paths from sapf will be used in case of timeout
15
```

Existing agents, new agents, new obstacles, the map, and the time step at which rerouting is to begin are input parameters. New obstacles are added to the map in line 2. In line 3, the conflict detection module is called and conflicts between new obstacles and existing agents are found. Note that at this moment there are no conflicts between agents, so conflict detection is likely very fast. Lines 4-10 find new paths for any agents in conflict with the new obstacles. If an agent's landing location is in conflict (in both space and time) with the new obstacles, its assigned a new landing location before a new path is

[1]Onishi, A. (2019) Rakuten package drone. (Photograph). Retrieved from https://asia.nikkei.com/Business/Companies/Rakuten-s-package-delivery-drones-to-take-flight-soon.

[2]Hussar fixed wing drone. (Photograph). Retrieved from https://www.upliftdronetraining.com/fixed-wing-drones/.

found (lines 5-7). If an agent's service location or current path is in conflict with the new obstacle, a new path is found using a sapf algorithm (lines 8-9). Agents without conflicts are ignored at this stage. After new paths for all *existing* agents are found, paths for all *new* agents are found, again using a sapf algorithm (lines 10-11).

At this point, a path has been found for all agents, however, these paths are not guaranteed to be conflict free. There may be newly created conflicts between agents. Thus, online independence detection is called with all agents as input. Recall that OID uses independence detection on top of an online mapf algorithm in order to quickly find conflict free paths for all agents. OID is inside a try catch block because of the possibility of timeouts occurring. As previously mentioned, if a timeout occurs during OID, the paths found using sapf are used.

When assigning a new landing location to an agent, the closest available landing location is used. This is done by performing a quick calculation using Euclidean distance based on the agents position at the rerouting time step. This can be improved, for example by use of an optimization algorithm with an objective function of minimizing the total distance between all agent's original landing location and new landing locations.

Internally, before any pathfinding is performed, an agents path is split into two separate paths. This split occurs at the rerouting time step. The part of the path that comes before the rerouting time step is not altered, only the part that comes after is changed. Once pathfinding is complete, the two parts are recombined back into a single path. A single agent's path is defined as a set of points in the map, $P = (p_s, p_{s+1}, p_{s+2}, \ldots, p_g)$. As an example, suppose an agent's path is as follows, $(p_s, p_2, p_3, p_4, p_5, p_g)$ and the rerouting time step occurs when the agent is between points p_3 and p_4. The agent's path is split into two, (p_s, p_2, p_3, p_{r-1}) and (p_r, p_4, p_5, p_g), where p_r is the position the agent occupies at the rerouting time step and p_{r-1} is the the position the agent occupies at the prior time step. Recall that π^i is the set of *conflict-free paths* for all agents, whereas P is a path for a single agent, not necessarily conflict-free. Section 4.1 outlines path implementation in more detail.

4 Implementation

This section provides implementation details and design decisions of the development work done for this thesis. All development was done in Java. JSON and serialized Java objects were used as the data exchange formats. The software I developed is part of an ongoing government contract between the laboratory at NII where I interned and NEDO and cannot yet be made publicly available. If it is possible, I will add the codebase to my public GitHub account (github.com/kymry) once the contract has concluded. The software is still being used by the team at NII and will likely be extended by future interns. The section is organized as follows: First, an overview of the map, agent and obstacle representation is given. Next, the strategies used for conflict detection are presented. The project was quite large and as such high-level software architecture design decisions had to be made. These decisions along with reasons for them are summarized. Finally, lower-level design decisions and implementation details of the mapf algorithms implemented and tested are provided.

4.1 Overview and Representation

Map Representation

The map in which pathfinding takes places is represented as a cubic grid in $\mathbb{R}^3$ composed of voxels with a 30 meter edge size. At each time step, an agent may either move to one of its 26 neighboring voxels (figure 7) or perform a wait move and remain in the same voxel. Although the map is defined as a grid of voxels, conflict detection does not use voxels. The voxels are simply used as a way to guide pathfinding. This will be further explained in the conflict detection section below (section 4.2).

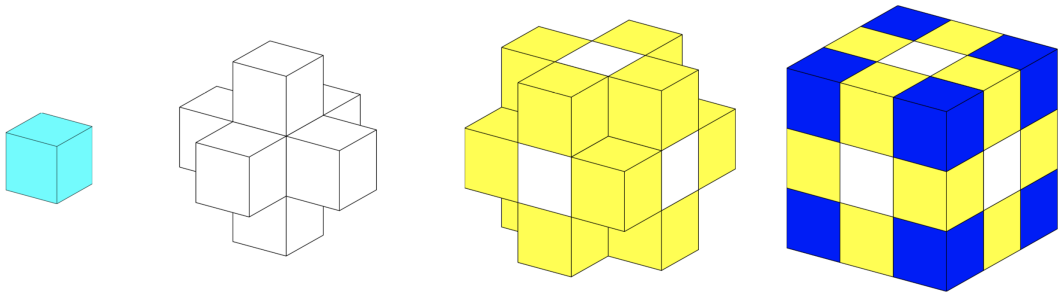


Figure 7: a voxel and its 6-neighborhood, 18-neighborhood, 26-neighborhood

The cubic grid is defined on top of the Universal Transverse Mercator coordinate system (UTM system). The UTM system is 2D coordinate system and therefore an additional altitude parameter is added (details to follow). The UTM system accomplishes the same task as GPS and was chosen due to its natural grid-based mapping and integer encoding, making it easy to conceptually interpret for graph-based search algorithms. It consists of a zone number, a hemisphere (Northern or Southern), an easting, and a northing. Zone number defines the geographic region — a total of 60 zones encompass the globe. Easting and northing represent the x and y coordinates within the zone, respectively. UTM coordinates are measured in meters and each zone has an arbitrarily chosen origin at (0,500000). For example, (43, 500432) means +43 meters in the y direction from the origin and +432 meters in the x direction from the origin. A point in the map is defined as a 4-tuple $p = (x, y, z, t)$ in $\mathbb{R}^4$, where x and y are as defined above, z is the altitude and t is the time step. Zone and hemisphere are not included because the scale at which UAVs operate is much too small for zones to have an impact on the end results. In order to speed up the search process, the map uses a 30 meter abstraction, meaning each UTM coordinate is scaled up by 30 meters, giving voxels a 30 meter edge length.

As mentioned above, the map is in 3D and uses UTM coordinates which represent actual geographic locations. Every location is at a different altitude from sea level and in order to accommodate this Google Maps Elevation API was used. This API takes a GPS coordinate and returns the altitude relative to sea level at that location. During pathfinding and collision detection, UTM coordinates are converted to GPS and used as input to the API. This was done in order to make the map as realistic as possible.

Agent, Path and Obstacle Representation

An agent is a sphere with a centroid, a radius, and a speed. Although the voxels have a 30 m edge length, the radius of an agent is not limited to 15 m and can vary among agents, as will likely be the case with real UAVs.

An agent position is its centroid, which is a point in the map as described above. Paths are defined as a set of agent positions. That is, a path is a set of points $(p_s, p_{s+1}, p_{s+2}, \ldots, p_g)$ where p_s and p_g are the start and goal positions. A pathline connects two agent positions and is defined as a pair of points (p_a, p_b) where p_a and p_b are any two *consecutive* points in an agent's path. Spherical agents were chosen to enable easy conflict detection and to simplify the representation of an agent internally (in Java classes).

Obstacles are represented as simplified convex polyhedra. Simplified in this case means two equal and parallel convex polygons separated by some height z. A set of tuples defines

the polygon $\langle (x_1, y_1), (x_2, y_2), ..., (x_m, y_m) \rangle$ where m represent the vertices of the polygon and a 4-tuple defines the other attributes $\langle z_{max} z_{min}, t_s, t_e \rangle$ where z_{max}, z_{min} define the maximum and minimum altitude of the obstacle and t_s, t_e are the start and end time steps during which the obstacle is active. Other objects such as buildings, trees and such are ignored. This geometrical representation (convex as opposed to concave polyhedra) was chosen because it provides relatively easy and lightweight object detection (in terms of development time in Java) using basic mathematics and it allows dynamic obstacles to be easily added to the map at any time. Figures 8 and 9 are visualizations of the geometric descriptions above.

To recap:

- point in map/agent position: $p = (x, y, z, t) \in \mathbb{R}^4$
- agent path: $P = (p_s, p_{s+1}, p_{s+2}, \ldots, p_g)$
- pathline: $pl = (p_a, p_b)$
- obstacle: $O = \langle (x_1, y_1), (x_2, y_2), ..., (x_m, y_m) \rangle$, $\langle z_{max}, z_{min}, t_s, t_e \rangle$

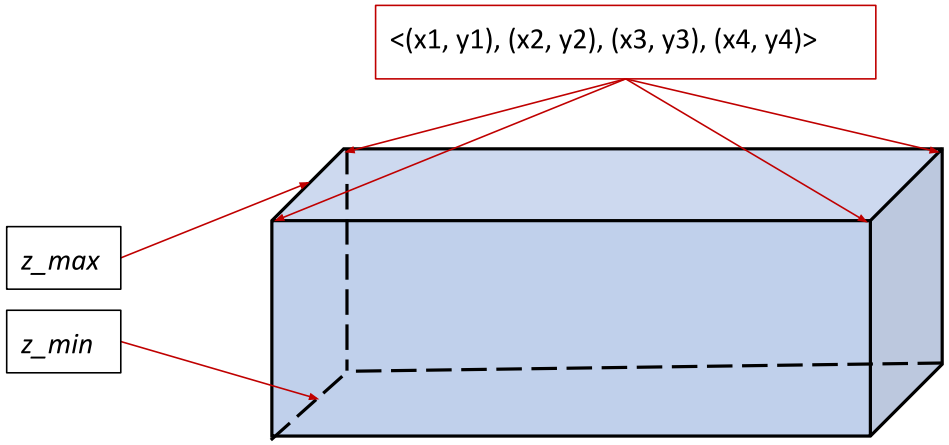


Figure 8: Convex polyhedron representing an obstacle.

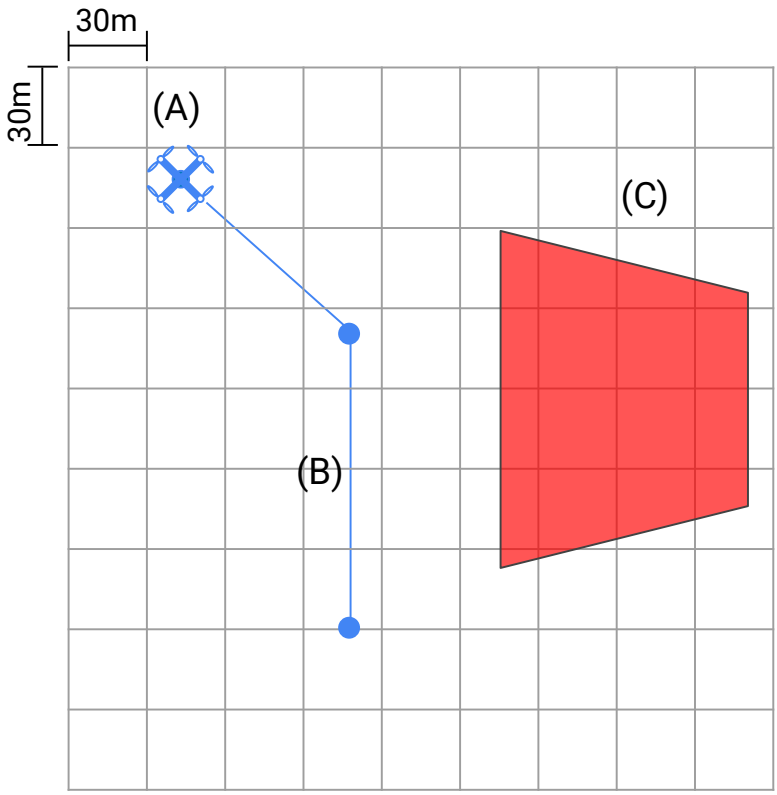


Figure 9: A 30x30 voxel grid (A) an agent and its path consisting of points in the map (B) a pathline connecting two consecutive agent path points (C): an obstacle represented as a polyhedron (only 2D shown here). Note that the obstacle does not correspond to voxels

4.2 Conflict Detection

This section provides a high level description of conflict detection between agents and obstacles and between agents themselves. Low level details of the conflict detection algorithms are not included but references are provided for those interested.

Agent-Obstacle Conflict Detection

Conflict detection between agents and obstacles is done using pathlines as described above. The exact area of the map an agent occupies at a given time step can be extrapolated from the pathline connecting two agent positions, the agent's speed, and the agent's radius. As previously mentioned, conflict detection is not done using voxels. It is done using geometry based collision detection algorithms. This method of conflict detection was simply a design choice. An equally feasible way of representing obstacles and performing conflict detection could be done by setting certain voxels (vertices in the graph) as occupied and prohibiting

agents from occupying those voxels.

When checking for collisions, the following steps are repeated for every pair of consecutive agent positions in an agent's path (p_s, p_{s+1}), (p_{s+1}, p_{s+2}), ... , (p_{g-1}, p_g).

1. Check if the pathline and obstacle are active at the same time step. If no, return false.
2. Check if the pathline is above or below the polyhedron with a distance greater than the agent radius. If yes, return false.
3. Check if either endpoint of the pathline is inside the polyhedron. If yes, return true. This uses the crossing number algorithm to check if the point intersects the polygons on the x,y plane, and then checks for intersection on the z-axis.
4. Check if the pathline intersects the polyhedron. If yes, return true.
5. Check the closest distance of the pathline to the polyhedron edges and if less than or equal to the agent radius, return true.
6. Check if the sphere created by the agent radius intersects the obstacle at either of the line endpoints. If yes, return true.
7. Return false.

Details of the geometry and algorithms used to perform the above steps are not included in this thesis. Many, if not all, of the main concepts and ideas can be found in [Ericson, 2004] and implementation examples can be found in [Sunday, 2012].

Agent-Agent Conflict Detection

As with obstacles, conflict detection between agents does not use voxels. The reason behind this stems from the wide open maps in which UAVs operate and the fact that agents can have varying radii. In voxel based collision detection, all voxels around two or more agents must be compared in order to detect collisions. Depending on the voxel size, this could be a large number of comparisons. Using geometric based collision algorithms, only a few calculations and comparisons need to be made. It is a four step process and is done between pairs of agents. This process is outlined below.

1. Check if agents are active at the same time. This is a quick check to rule out pairs of agents that cannot possibly be in conflict. If no, return false.
2. For every pair of agent pathlines, check if they overlap in time.

3. For every pair of pathlines that overlap in time, check if they intersect in space.
   (a) Calculate the minimum distance between the pathlines.
   (b) If the minimum distance is less than the sum of the agents radii, there is a *possible* conflict between the agents.
4. For all pairs of pathlines that have a *possible* conflict, check for definitive conflict using the Closest Point of Approach (CPA) algorithm.
   (a) Cut the pathlines into smaller segments based on time step (e.g. 1 second segments).
   (b) For each pair of segments, check if they overlap in time.
   (c) If so, check for definitive conflict using the closest point of approach (CPA) method.

In this approach, the first three steps can very quickly determine if a pair of agents have a *potential* conflict between them, and if so, where in space and time that conflict may occur. The fourth and final step then *definitively* determines if there is a conflict. This multi-step approach was taken because UAVs typically operative in a wide-open map, making conflicts rare. If conflicts are rare and UAVs operate far from each other in space, the first three steps are sufficient to determine if two agents cannot be in conflict with just a couple of quick constant time comparisons. The alternative would be checking voxel by voxel to search for conflicts, which is a huge waste of time for agents operating far from each other in space.

Details of the CPA algorithm can be found in [Ericson, 2004] and [Sunday, 2012]. In short, CPA takes two points in space and uses their velocity vectors to compute the points along their paths at which they are closest to each other in space and time. Once the closest point of approach is determined, collisions are detected using the agent's radii. This check is necessary, as the minimum distance between the two agent paths is unlikely to be the same as the closest point of approach.

4.3 Software Architecture

This was quite the large development project, so my team and I at the National Institute of Informatics decided to use objected orient programming techniques and design the software in a modularized way. There are three modules and each was developed such that it could be used standalone or in combination with any other module. Each can be

treated as a black-box in which algorithm choices, data format output types, and other various choices are parameterized. We decided to modularize the project code base for a couple of reasons.

- Breaking up the project into more manageable pieces allowed for a simpler and cleaner development approach and allowed for more thorough testing to be performed throughout the development process.
- One of the goals of this project was to test different path finding algorithm combinations in order to compare performance and determine which combinations are well-suited for the the wide open maps used by UAVs. Having a standalone module for algorithm selection made this process straight forward.
- In order to better analyze the general characteristics of conflicts in the UAV state space, to determine which types of conflicts occurred, and to see the impact that no-fly zones have on UAVs, we needed an easy way to detect conflicts without simultaneously finding new paths. A standalone conflict detection module allowed us to do this.
- Rerouting is a special case of path finding, but has quite significant changes and will not always be used. Therefore, we decided it best to keep it as its own module which can be used when needed.

The three modules are "conflict detection", "pathfinding", and "rerouting", and are briefly described below.

Conflict Detection Module

As the name implies, the conflict detection module detects conflicts both between agents themselves and between agents and obstacles. The input to this module is quite simple and consists of the map, list of all agents and their associated paths, and a boolean trigger for activating the path finding module. The output consists of a set of conflicts and the associated time step. A conflict is defined as a custom data type that includes instance fields such as agent(s) id, obstacle id, time step, location, type of conflict, etc. These conflicts can then be passed to the path finding module if desired.

Pathfinding Module

This module performs single-agent and multi-agent pathfinding. Given a set of agents and a map, it finds conflict free paths for all agents. Throughout the path finding process,

periodic calls to the conflict detection module are made. Any valid combination of single-agent and multi-agent path finding algorithms can be provided as input. Prior to execution, a check is performed to ensure the provided combination is valid, as not all algorithms are compatible. Any number of conflicts and/or static obstacles can be provided as input. Static obstacles are added to the map prior to path finding.

Rerouting Module

This module performs the dynamic inflight rerouting. It makes frequent calls to both the conflict detection and path finding modules. This module takes the same input as the path finding module along with a few additional parameters including the current time step, list of new no-fly zones, time step at which rerouting will occur and a list of newly added landing locations.

4.4 Pathfinding Algorithm Implementation Details

This section gives implementation details of the pathfinding algorithms, discusses some design decisions, and presents two custom data structures that were developed in order to improve the pathfinding algorithms.

One important facet of the agent representation is that agents may have varying speeds. CBS and its variants can handle this by default without issue, since single agent A* (or a variant) is used at the core and the conflict detection described above elegantly handles the varying speeds. The only algorithm that was adapted to handle varying speeds was A* with Operator Decomposition. Details are discussed in the corresponding section below.

Best-first Search

Best-first search (BFS) was implemented as an abstract class, allowing it to be extended by any A* based search algorithm. A thread pool was used internally to allow single and multi-agent search algorithms to be executed and controlled independently. The main purpose of this was to allow a search to be terminated when a given time limit was reached. It also allows multiple searches to be run simultaneously. The Java min/max priority queue class was used for the open list.

Pathfinding algorithms that utilize BFS use heuristics to guide the search and because heuristics vary among algorithms, separate mapf and sapf heuristic interfaces were implemented and referenced in the BFS class. Any class that extends the abstract

BFS class can then supply their own heuristic, as long as it implements the interface. Implementing the heuristics as an interface also allows new heuristics to be added without changing the BFS code.

Independence Detection

Independence detection (ID) was chosen because it significantly improves the running time of many mapf algorithms and the rerouting process requires it as discussed in section 3.3. It was developed as a class with no dependencies in the pathfinding module so that it could be used as a framework on top of any mapf algorithm. In order to do this, ID was developed around the command design pattern.

The command pattern encapsulates all of the information needed to perform an action or to trigger an event in an object. In other words, it allows methods to be easily passed around a program to be used in a decoupled manner by disparate classes and threads. ID is not a pathfinding algorithm itself, the caller needs to provide the algorithm. There are two ways this could be done. A list of pathfinding algorithms could be made available and the caller then specifies which algorithm they want to use. This has the limitation that the caller has a limited choice of algorithms. The other option is to allow the caller to provide their own pathfinding algorithm that implements an interface. This is exactly what the command pattern does and provides unlimited flexibility on the callers part. Figure 10 is a UML diagram of the command pattern. The ConcreteCommand in this case is the pathfinding algorithm and is provided by the Client. Command is an interface that ConcreteCommand implements, allowing any algorithm to be used. The Caller (same as client in this case) initiates the action (pathfinding) request. The Receiver performs the independence detection by utilizing the ConcreteCommand object (pathfinding algorithm) provided by the Client.

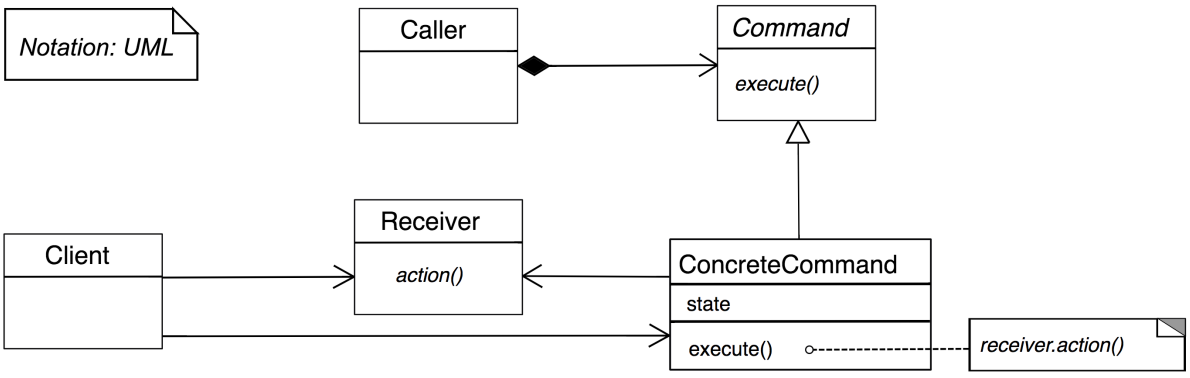


Figure 10: Command Design Pattern

The main idea behind ID is to separate agents into disjoint sets. Doing this requires a data structure that allows elements to be added and removed from sets and to determine which set contains an element(s) quickly. Java provides a few different set based data structures that provide $O(1)$ add, remove and contains operations. This however is not sufficient for ID due to the fact that determining which set contains an element is still linear in the number of sets, which is the number of agents k in this case. To overcome this deficiency a custom disjoint-set forest was developed [Cormen et al., 2009].

A disjoint set forest is a set of rooted trees. Each tree is a disjoint set and each node is a single element, agents in this case. Initially, each element is assigned to its own tree and acts as the root. Internally, the disjoint set forest contains four data structures all based on hash maps: *parent*, *values*, *optimal cost*, and *rank*. *Parent* identifies the group the agent is in. *Values* is a mapping of agents to a list of other agents that are in the same group and is an auxiliary feature that allows one to quickly identify all agents contained within a group without preforming a tree traversal. *Rank* is the depth of the tree. *Optimal cost* is the total path cost of all agents in the group. When a conflict is found during ID, new paths are found for all agents in a group. The cost of these new paths is required to be equal to the cost of the old paths. Optimal cost allows this check to be preformed in constant time.

There are three main functions in the disjoint-set forest: *make_set()*, *find_set()* and *union()*. *make_set()* is used just once during the initialization phase and creates a new set for each element. *find_set()* determines which group an element is in. *union()* combines to groups. A standard disjoint-set forest has a $O(1)$ *union()* operation and $O(d)$ *find_set()* operation where d is the depth of the tree, therefore determining if two agents are in the same group is $O(2d) = O(d)$. In order to improve this, two heuristics are added: union by rank and path compression. Union by rank is used during the *union()* procedure and makes the root of the tree with the smaller rank point to the root of the tree with the higher rank, reducing the average size of the trees. Path compression is a recursive algorithm that executes during the *find_set()* operation by forcing each node in the tree to point to its parent, thus reducing the tree to a depth of 2. Used in combination, the total running time has been shown in to be $O(m)$ where m is the total number of *make_set()*, *find_set()* and *union()* operations [Cormen et al., 2009].

Conflict Based Search and Suboptimal Variants

Conflict based search was implemented as a concrete base class, allowing it to be extended by GCBS and ECBS. CBS was primarily chosen because of its favorable trade-off between

solution quality, execution time and ease of implementation. As shown in [Sharon et al., 2015] and [Barer et al., 2014] CBS and its variants either outperform or perform nearly as well as other state of the art solves such as ICTS, M* and MDD-SAT. CBS however is significantly less complicated to implement, allowing for quicker development time and testing. Both GCBS and ECBS were chosen because they naturally extend and decrease run time over CBS.

As mentioned earlier, GCBS is a greedy version of CBS that adds heuristics to the high and/or low level searches. It has been shown that in high level search, using a combination of heuristics in a round robin fashion improves running time more than any single heuristic [Standley, 2010]. For the low level search, nodes with fewer conflicts are preferred even at the expense of optimality. The following three conflict heuristics were used for the high level search.

- number of conflicts — count of conflicts encountered in a CT node
- number of pairs — count of the number of pairs of agents with at least one conflict between them
- number of conflicting agents — count of agents that have at least one conflict

These heuristics were implemented as comparators in Java, allowing them to be used in min/max priority queues. This leads to a problem when using 3 heuristics in a single search, as standard priority queues only accept one comparator, whereas three are needed. One possible solution is to use 3 separate priority queues - one for each comparator. This is far from ideal as each time a node is polled from one priority queue, that same node must be found and removed from the other two queues, requiring $O(n)$ for each queue just for retrieval. $O(n)$ is required because Java uses a heap for priority queues and thus the node to be deleted can only be located by iterating over every element in the queue. Furthermore, implementing three priority queues separately in a best first search is messy and error prone.

To get around this, a custom data structure I'm calling a Three Priority Queue was developed. A three priority queue is a generic class that accepts any three comparators upon creation (note: it is possible to extend this to an arbitrary number of comparators). Inside are three Java TreeSets, one for each comparator. A TreeSet in Java is a priority queue that uses a self-balancing binary search tree to store elements. A few notable properties include:

- is a set, so duplicated values are not allowed
- stored in a self-balancing binary tree structure

- remove/add/poll/contains is $O(\log(n))$

Inside the three priority queue is an iterator that iterates over the three priority queues in sequential fashion. Every time the queue is polled, the iterator advances and polls the next TreeSet while simultaneously deleting the corresponding node from the other two TreeSets. This structure enables the process of using three queues to be safely encapsulated.

A* with Operator Decomposition and Independence Detection

A* with operator decomposition and independence detection (A*+OD+ID) was chosen because it is a *lighter* algorithm and would be interesting to see how it compared to the more complex CBS and its variants. Lighter meaning conceptually simpler and requiring less memory. If a simpler and easier to implement algorithm performs just as well as a more complex one, the simpler one will almost always be chosen. Furthermore, A*+OD+ID doesn't seem to have been tested in the wide open map that UAVs operate in, providing a good opportunity for research.

At first, standard A* with the Euclidean diagonal distance heuristic was used. However, this quickly proved to be inadequate. Due to the wide open map in which UAVs operate, finding a path for even a single agent sometimes took tens of seconds or even minutes depending on the length of the path. After some investigation, it was found that this was occurring in two specific situations.

1. When an agents goal location could not be reached by following a non-diagonal, straight line path from the start location (which almost always occurs). In this situation, many same cost paths exist and A* searches many of them before finally reaching the goal position. This occurred due to using the diagonal distance heuristic in conjunction with a voxel based search. A* extends the BFS class described earlier, which utilizes a Java priority queue internally to store the open list of nodes. When two elements with the same value are inserted in a Java priority queue, ties are broken arbitrarily, thus causing many same cost paths to be searched before reaching the goal.

2. When a large obstacle was placed in between an agent's start and goal location. In this situation, A* will work as designed and go straight towards the goal until it hits the obstacle. It will then try many paths between the start location and the obstacle before finally finding the edge of the obstacle and going around.

Figure 11 is a simplified visual representation of theses two cases. The lighter grey is the voxels that A* may check during its search and the black is the final path. As can be

seen, A* searches many more voxels than is necessary, significantly slowing the search. One can imagine in a large map, the number of light grey voxels is prohibitive.

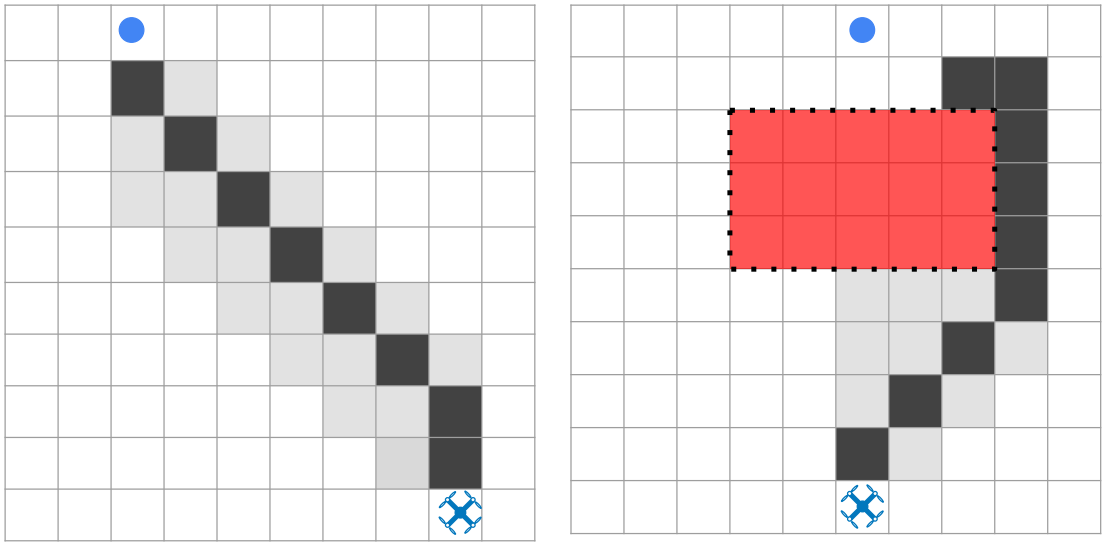


Figure 11: Left: A* issue 1. Right: A* issue 2

To solve the first situation, a weight, w, was added to the heuristic, changing the cost function to $f(n) = w \cdot h(n) + g(n)$. This makes A* suboptimal, bounded by the weight, but improves running time significantly. This was deemed an acceptable trade-off. A solution to the second situation was not implemented due to time constraints. One possible solution is to use FA-A*, a focal any angle variant of A* presented in [Cao et al., 2017]. FA-A* utilizes a visibility graph to decompose the map into a set of candidate nodes, which are the corners of geometric obstacles. The true shortest path of an agent only changes directions when an obstacle is encountered (assuming single agent search and continues state space). Therefore, an agent will pass through one of the candidate nodes if an obstacle is present. In the case of UAVs, there are few obstacles in a wide open map. The thought is that because there are so few obstacles, the candidate node list will be small and a path will be found quickly.

A*+OD as presented in [Standley, 2010] and described in section 2.4 seems to assume that all agents begin at the same time step. It also implicitly assumes that when an agent is in a goal location, another agent cannot occupy that same location. These assumptions do not hold with the problems surrounding UAVs addressed in this thesis. One idea surrounding UAVs is the idea of a drone port. A drone port is simply a specified location in which a UAV is allowed to enter and exit the airspace. This idea of a drone port is

exactly the same as an airport, in which planes take off sequentially, not simultaneously. Furthermore, when a UAV exits the airspace and lands in a drone port, it effectively disappears from that location, allowing another UAV to occupy that same location shortly after. This introduces two requirements that A*+OD must be adapted to handle (1) agents may begin their paths at different time steps (2) agents disappear from their goal locations, allowing other agents to occupy the location shortly after.

A*+OD advances agents one at a time, selecting a move for a single agent at each intermediate node. The ordering of agents is unimportant, therefore agents were arbitrarily ordered by agent id. Internally, each agent has start and end time parameters. The start time is fixed. At each step of the algorithm, before a move for an agent is chosen, the current time step is compared to the agent's start time parameter. If the start time is less than or equal to the current time step, a move is selected for the agent, otherwise the agent is ignored and execution moves to the next agent - a new intermediate node is not created. This accommodates requirement 1. Figure 12 displays this idea. The top bar is the current time step during search. Only agents that are active during the current time step are included in the search. For example from $t = 0$ to $t = 5$ paths for agents ab are being found. Then from $t = 5$ to $t = 15$ paths for agents a, b, c are being found.

This algorithm deals with varying agent speeds in a subtle way. Each time a node is expanded and an agent selects a new move, the agent can either move into one of its 26 neighboring voxels or remain in the same voxel. If it decides to move into a neighboring voxel, the time step associated with this move is updated by calculating the amount of time it takes the agent to arrive there, given its velocity and previous location. This updated time is then used as the time step associated with the new node created with this move. What this means is that this implementation of A*+OD doesn't follow discrete time steps as a typical implementation might. Depending on the problem at hand, this may or may not be adequate. In the case of UAVs, this introduces a notion of fairness, ensuring the UAVs are able to progress through the map to their goals as if they had the same speed. This was a design choice in order to treat all UAVs equally.

When an agent reaches its goal, t time units are added to the end time parameter to allow the agent adequate time to be removed from the location. This is equivalent to a UAV being removed from a drone port upon landing, freeing up the space for another UAV to land. These t time units effectively act as wait moves. Normal conflict detection can then be used by other agents to detect if the location is occupied. This accommodates requirement 2.

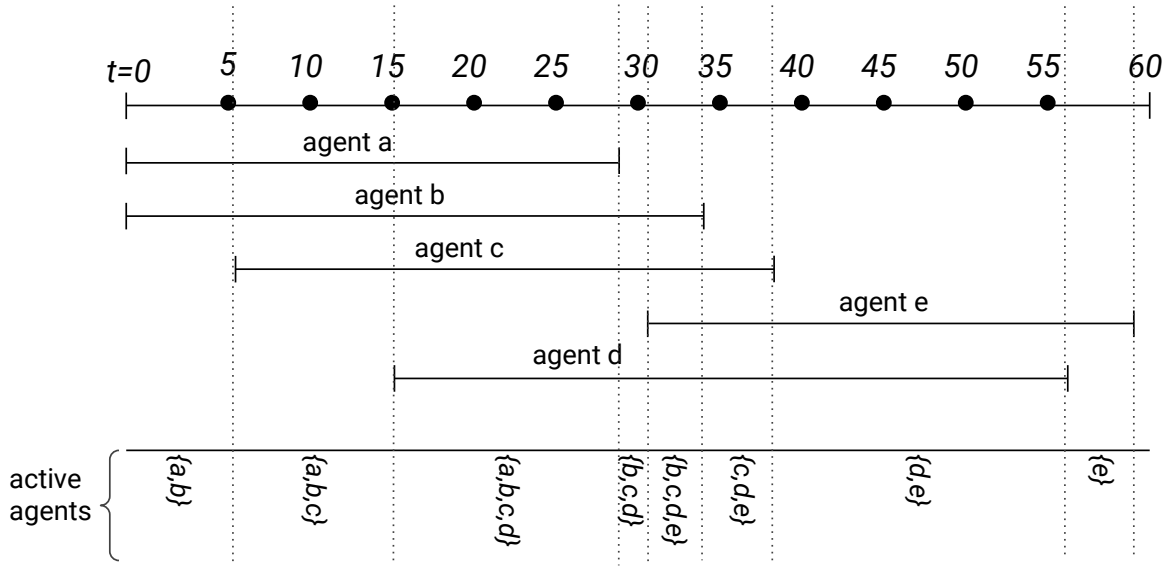


Figure 12: A*+OD search

5 Experimentation and Results

This section is organized into three parts. The first part describes the simulations used for testing, including the data, map structure, agent characteristics, and operation types.

The second part aims to prove that fast, inflight rerouting algorithms are a necessity for UAVs. The computational performance of the inflight rerouting algorithm (UAVReroute) developed for this thesis is also tested and evaluated. The goal here is to show that this algorithm is a good starting point for the development of more robust inflight rerouting systems that can be used in the industry — a proof of concept.

The third part tests and compares the performance of a few mapf algorithms applied to the maps in which UAVs operate, with a goal of determining which perform best when applied to UAVs. As mentioned in the introduction, there has not been much testing of mapf algorithms applied to maps in 3D space, leaving the correct choice of algorithms unclear. This is the reason why the maps used for testing in this thesis are quite different from those used in testing the state of the art. Due to time constraints, only a few mapf algorithms were tested, but nonetheless, some good data was obtained and analyzed.

5.1 Simulation Overview

External Data

The simulations used to perform the testing were made to be as realistic as possible in order to ensure the results were meaningful and could be analyzed and referenced by future researchers. The laboratory where I performed these tests contracted with a third party consulting firm to gather data about the future scale and usage of UAVs for a city in Japan in 2030. This is part of an ongoing project with NEDO, so the firm name and the comprehensive data set is not publicly available. The data obtained considered various businesses delivering goods (e.g. package delivery to customers or blood delivery between hospitals) and provided average number of daily UAV operations throughout the city and average distance per operation. Recall that an operation is one flight of a UAV — in this case that is one full delivery.

The summarized data obtained from the consulting firm is as follows:

- City (map) size: 14 × 17 kilometers
- Operation distance: 600 — 4000 meters

- Maximum number of daily operations (13 hour day): 21,235
- Minimum number of daily operations (13 hour day): 13,910

It should be emphasized that this is simply one study performed by a consulting firm that applies to one city in Japan. Nevertheless, it is a good starting point and was used as a general guide for the testing done for this thesis. The operation distance was roughly followed for all simulations.

Map

The simulations were performed on a 10 km x 10 km x 150 m map consisting of approximately 5% static obstacles representing no-fly zones (e.g. airports, police stations, hospitals, etc.) placed uniformly at random in the map. The map boundaries in UTM coordinates are: northwestern corner (482604, 4245254), southeastern corner (492604, 4235254). These coordinates are in Zone 54 S. The placement of obstacles in the map is not important, as every city and location will likely have a different arrangement of no-fly zones. The density of the obstacles on the other hand is important, as the map in which UAVs operate will likely be very open. Recall from section 4.1 that a static obstacle is represented as a convex polyhedron. The obstacles occupy the entire 150 m altitude of the map, meaning UAVs cannot fly above or below the obstacle. UAVs are a low altitude aircraft, thus a 150 m altitude limit was used in order to make the simulations more realistic. A 30 meter voxel abstraction was used, meaning the map consisted of roughly $333 \times 333 \times 5$ voxels.

Agents

The agents are considered to be 20 meters in diameter. Even though UAVs themselves are quite small, a 20 meter diameter was used to account for positioning error and as a way to enforce a minimum distance between obstacles and other UAVs. Positioning error is the difference between a UAVs true location and its recorded location. GPS inaccuracy, high-winds blowing a UAV slightly off course, statistical noise, faulty internal computer and many other factors may cause positioning errors. Although positioning error is not present in the simulations created for this thesis, the 20 meter diameter was included in order to make the simulations as realistic as possible. The agents were all set to have a constant velocity of 15 m/s.

Data Generators

Recall from section 3.2 that an operation consists of a start location s, task location l and goal location g. For the simulations, an operation was effectively split into two separate pathfinding problems, one for the path from s to l and one from l to g. This was done because tasks take varying amounts of time and therefore the time a UAV spends at l varies by operation. Another possible approach is to force wait moves corresponding task duration in which an agent remains in the same location for a number of time steps, but this seemed overly complicated and is not what pathfinding algorithms are designed for.

In order to generate s, l and g, two different random number generators were created.

Generator A takes as input a pair of integers representing the maximum and minimum easting or northing in the UTM coordinate system. See section 4.1 for an an overview of the UTM system. The output is a number chosen uniformly at random between these two integers, representing either the northing or easting. To generate the start location s, this generator is called twice, once for the northing and once for the easting.

Generator B takes as input two pairs of integers. The first pair is a UTM coordinate representing a point in the map consisting of an easting and a northing. The second pair represent a minimum and maximum distance from the UTM coordinate. The generator then returns a point uniformly distributed between the two concentric rings created using the UTM coordinate as the center and the min and max as the radii. For example, in order to generate the task location l, the start location s, along with a min and max radius is used as input.

Both of these generators simply generate points distributed uniformly at random within a given area of specified size. Generator A is used for rectangular area and generator B is used for circular area. They can be easily replicated.

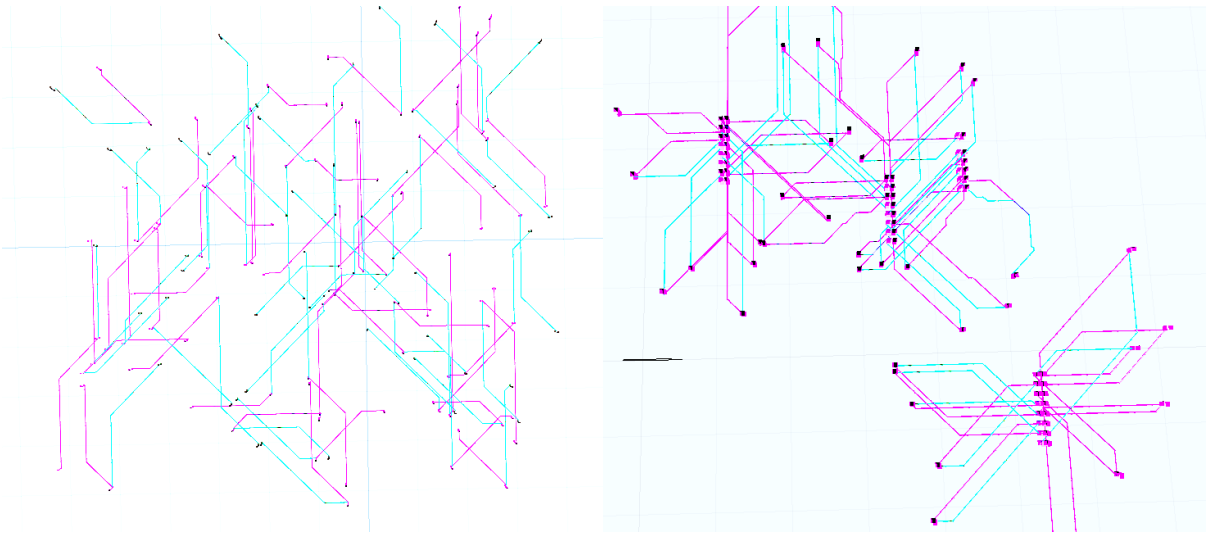
Operation Types

Two different types of operations were simulated and tested: *open map operations* and *drone port operations*.

Drone port operations have the same start and goal location in a predefined drone port and a task location 1000 — 2000 m away chosen using generator B. The drone ports themselves were placed uniformly at random within the map using generator A. Recall from section 2.7 that a drone port is just a specified location in which a UAV is allowed to enter and exit the airspace.

Open map operations consist of start, task and goal locations placed uniformly at random in the map. The start location was chosen using generator A and the task and goal locations were chosen using generator B. The distance between each of s, l and k ranges from 1000 — 2000 m.

Figure 13 shows the structure of the drone port and open map operations. (a) are open map operations that are placed uniformly at random throughout the map. (b) are drone port operations. There are four drone ports in this image, each with multiple operations.



(a) Open Map Operations **(b)** Drone Port Operations

Figure 13

5.2 Inflight Rerouting Algorithm Tests

For all tests, 25 independent simulations were run and the average was taken.

Inflight rerouting testing was done with both types of operations. For the drone port operations, 15 ports were distributed uniformly at random around the map using generator A, each allowing 10 UAVs to takeoff and/or land simultaneously. The number of simultaneous operations ranged from 100 — 200 for drone port operations and 100 — 800 for open map operations.

A simulation works as follows; First, generate operations and find optimal conflict free paths using CBS. Next, fill roughly 5% of the map with dynamic obstacles uniformly distributed throughout the map. Finally, using a predefined rerouting time step, perform the rerouting using the UAVReroute algorithm. Internally, OID, which uses independence detection on top of A* for sapf and ECBS with Focal A* for mapf was used. A weight of 1.5 was used for ECBS.

Figures 14 and 15 display the number of UAV conflicts caused by the dynamic obstacles being added to the airspace. Figure 14 corresponds to the drone port operations and Figure 15 corresponds to the open map operations.

- **No rerouting** means no rerouting of any kind was performed. Conflicts in this case is the number of UAVs in conflict with the dynamic obstacles.
- **Single-agent pathfinding only** means rerouting was performed using only A* for single agents. Conflicts is the number of UAVs that are in conflict with the dynamically added obstacles or with other UAVs after the rerouting was performed. If a UAV has both types of conflicts, it was not counted twice.
- **Multi-agent pathfinding** means rerouting using the UAVReroute algorithm. Again, conflicts is the number of UAVs that are in conflict with the dynamically added obstacles or with other UAVs after the rerouting is performed.

Tables 1 and 2 display the same information, along with the running time measured in seconds and average path length in meters. Operations is the number of simultaneous operations, A* is rerouting using only single-agent A* and UAVReroute is rerouting using the UAVReroute algorithm.

As can be seen, the number of conflicts remaining after UAVReroute is fewer than with single-agent A*, but it is not zero. This happens when a path cannot be found within the given time limit. A timeout of 1 second was used for A* and Focal A*, which in theory should be plenty of time to find a path for a single agent. After some investigation, it was found that A* was failing in two cases.

- When a dynamically added obstacle was very large and an agent was near the center of the obstacle at the rerouting time step, A* often timed out. This happened because the agent could not leave the obstacle in its desired direction before the obstacle became active, forcing A* to search many different paths in order to find an exit.
- As mentioned in section 2.1, when an agent's path went directly through a large obstacle, A* often timed out trying to find a path around the obstacle. Future inflight rerouting systems will need to take these two limitations into consideration.

Although some conflicts remain, we can see that the number is reduced after rerouting with UAVReroute. This explicitly shows the necessity for sophisticated inflight rerouting algorithms as even a single conflict between UAVs is unacceptable. It is notable that both A* and UAVReroute have similar running times. This is likely due to a couple of reason. In order to find initial paths for agents, UAVReroute runs A* for all agents in conflict

before running ECBS. From Figure 14 and 15 it seems that there are not many conflicts between agents after single agent A* is run, especially when the number of operations is low. This means that independence detection can sometimes solve most conflicts without calling ECBS and if ECBS is called, it is likely quick to solve the few conflicts that remain. This shows that UAVReroute is quite powerful and might be a good algorithmic choice for inflight UAV rerouting systems.

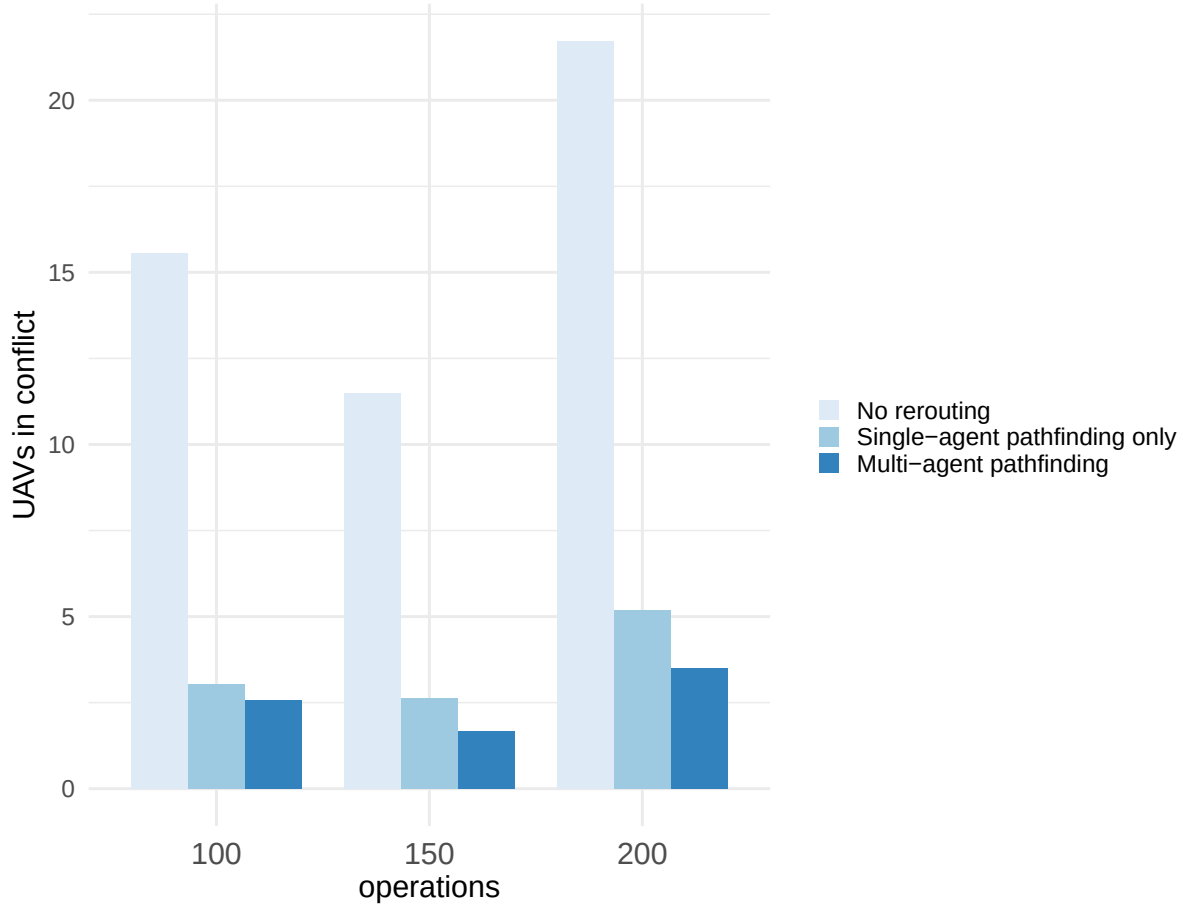


Figure 14: Inflight rerouting conflicts — drone port operations

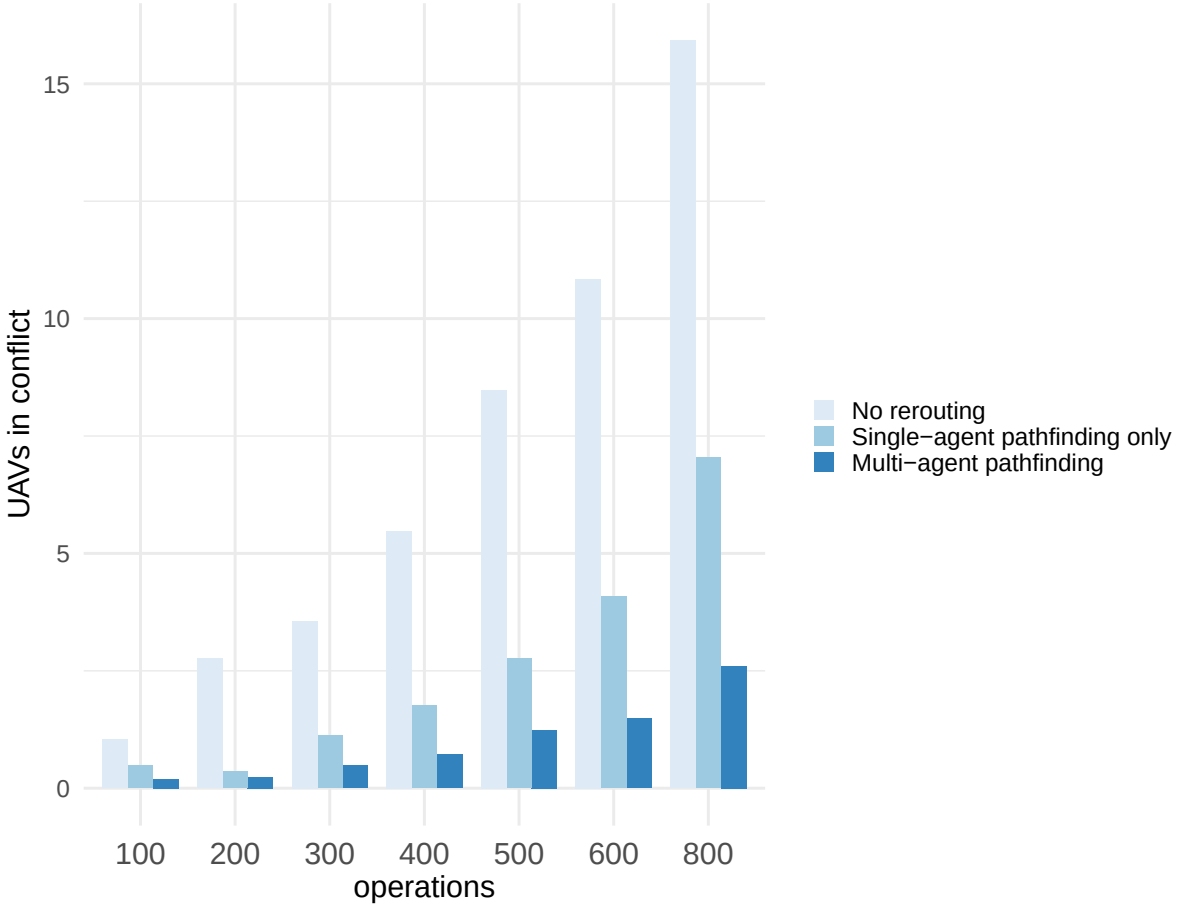


Figure 15: Inflight rerouting conflicts — open map operations

	Running Time (s)			Path Length (m)			Conflicts Before		Conflicts After	
Operations	A*	UAVReroute	ratio	A*	UAVReroute	ratio	A*	UAVReroute	A*	UAVReroute
100	3.6	3.5	0.97	1478	1510	1.02	15.6	15.6	3.0	2.6
150	4.4	4.9	1.13	1516	1539	1.00	11.5	11.5	2.6	1.7
200	8.0	8.2	1.01	1496	1501	1.00	21.7	21.7	5.2	3.5

Table 1: Dynamic inflight rerouting — drone port operations

	Running Time (s)			Path Length (m)			Conflicts Before		Conflicts After	
Operations	A*	UAVReroute	ratio	A*	UAVReroute	ratio	A*	UAVReroute	A*	UAVReroute
100	0.7	0.6	0.87	1527	1539	1.01	1.0	1.0	0.5	0.2
200	0.8	0.9	1.05	1521	1534	1.01	2.8	2.8	0.4	0.2
300	3.2	2.8	0.89	1537	1550	1.01	3.6	3.6	1.1	0.4
400	4.6	4.9	1.06	1522	1534	1.01	5.5	5.5	1.8	0.7
500	6.4	7.3	1.14	1527	1541	1.01	8.5	8.5	2.8	1.2
600	6.7	8.1	1.22	1521	1533	1.01	10.8	10.8	4.0	1.4
800	12.9	14.6	1.13	1521	1533	1.01	15.9	15.9	7.0	2.6

Table 2: Dynamic inflight rerouting — open map operations

5.3 Multi-Agent Pathfinding Tests

Multi-agent pathfinding testing was done using open map operations. The number of simultaneous operations ranged from 100 — 800. A*+OD+ID, CBS, GCBS and ECBS were experimentally tested. The heuristics used for GCBS were number of conflicts, number of pairs of conflicts and number of agents in conflict.

A*+OD+ID was experimentally tested, but the results were not included. With the large number of agents used in testing, A* with operator decomposition was no where near fast enough. As shown in [Felner et al., 2017], it was known that A*+OD alone was not fast enough. The thought was that by adding independence detection on top of A*+OD, the running time would be reduced enough to provide good results for use in dynamic inflight rerouting algorithms. In general, this was true; independence detection allowed A*+OD to be run with only one or two agents. Unfortunately, in basically all simulations, there were a couple of instances in which many agents were grouped together, causing A*+OD to be unusable due to long running times.

Looking at figure 4, the first thing to note is the path length, corresponding to the solution quality, is basically the same for all three CBS variations. Plus or minus a few meters for a UAV is negligible and can be safely ignored. The running time between the optimal CBS and suboptimal ECBS is as expected, with CBS being slower. However, GCBS is unexpectedly the slowest, when it should be quite fast. With the open map used by UAVs, there are not many conflicts and the conflicts that do exist are usually solved quite easily by finding an alternate path of the same length. The strength of GCBS only arises when there are many conflicts between agents that are difficult to solve, which is not the case here. However, it is still odd that GCBS is the slowest at times. I speculate that this is primarily due to implementation details, but the internal three priority queue structure used by GCBS to store the nodes for the three heuristics may also contribute. GCBS is doing extra work each time a node is expanded during search because it must update all three priority queues, costing $O(log(n))$ each time.

We can conclude that either CBS or ECBS is a good candidate for mapf applied to UAVs. We can see that there is not much of a trade off between bounded suboptimal and optimal algorithmic solutions, at least in this case. With this, we can hypothesize that other optimal mapf algorithms may be adapt well to UAVs due to the open maps with few obstacles. This provides a good direction for future testing and research.

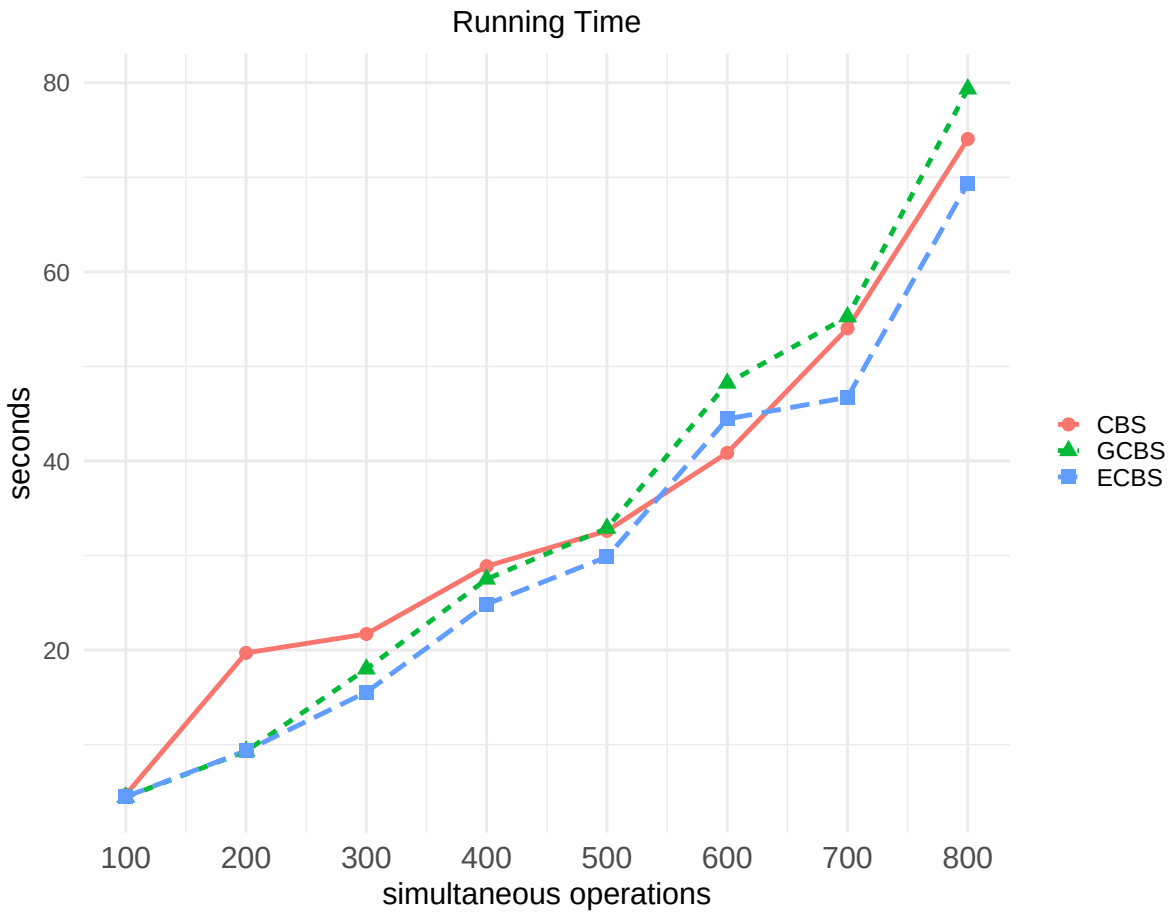


Figure 16: MAPF Running Time

	Running Time (s)			Path Length (m)			Path Length Ratio		
Operations	CBS	GCBS	ECBS	CBS	GCBS	ECBS	CBS	GCBS	ECBS
100	5	4	4	1890	1890	1890	1.0000	1.0000	1.0000
200	20	9	9	1887	1887	1887	1.0000	1.0000	1.0001
300	22	2	16	1887	1888	1887	1.0000	1.0003	1.0000
400	29	28	25	1883	1884	1884	1.0000	1.0005	1.0005
500	33	33	30	1882	1883	1882	1.0000	1.0000	1.0000
600	41	48	44	1883	1883	1884	1.0000	1.0001	1.0005
700	54	55	47	1883	1884	1884	1.0000	1.0004	1.0005
800	74	79	69	1879	1880	1882	1.0000	1.0005	1.0010

Table 3: Mapf — open map operations

	Running Time (s)			Path Length (m)			Path Length Ratio		
Operations	CBS	GCBS	ECBS	CBS	GCBS	ECBS	CBS	GCBS	ECBS
100	5	4	4	1890	1890	1890	1.0000	1.0000	1.0000
200	20	9	9	1887	1887	1887	1.0000	1.0000	1.0001
300	22	2	16	1887	1888	1887	1.0000	1.0003	1.0000
400	29	28	25	1883	1884	1884	1.0000	1.0005	1.0005
500	33	33	30	1882	1883	1882	1.0000	1.0000	1.0000
600	41	48	44	1883	1883	1884	1.0000	1.0001	1.0005
700	54	55	47	1883	1884	1884	1.0000	1.0004	1.0005
800	74	79	69	1879	1880	1882	1.0000	1.0005	1.0010

Table 4: Mapf — open map operations

6 Conclusion and Future Work

This section lists my contributions, summarizes the outcomes of this thesis and discusses possibilities for future work.

6.1 Contributions

Individual contributions

1. All multi-agent pathfinding an online mapf research, analysis and discussion.
2. Design, development, testing and analysis of all experiments.
3. The entire dynamic inflight UAV rerouting algorithm (UAVReroute), including research and development.
4. Development of and design decisions for: independence detection framework, operator decomposition framework, A*+OD+ID, online mapf solution, GCBS, weighted A*.

Collaborative work (improvements made on work from lab team and previous intern)

1. Development of map, agent, obstacle representation.
2. Development of CBS, ECBS.
3. Design decisions for high level software architecture.
4. Collision detection.

Basically, all research for both mapf and online mapf was done independently. Testing was as well. A basic working version of the core software was built by the lab team and a previous intern. Using this, I made many necessary improvements and changes to allow for modularity of the codebase, faster execution, ease of future development should another intern continue the work, improved and more thorough testing using unit tests, numerous bug fixes, high and low-level design changes, changes to agent representation, etc.

6.2 Conclusions

It seems inevitable that in the near future UAVs will become commonplace and used by governmental organizations and companies to perform various tasks. As the scale of UAV operations continues to increase, so will the need for efficient mapf algorithms to

find conflict-free routes. In addition, a new method of contingency planning for when an unexpected event occurs in the airspace and UAVs are forced reroute inflight must be developed. With that said, this thesis had three main goals: (1) Thoroughly examine state of the art mapf algorithms in order to determine which perform best when applied to UAVs (2) Experimentally test a few mapf algorithms in the 3D maps used by UAVs (3) Design, implement, and test a dynamic inflight rerouting algorithm to handle no-fly zones being dynamically added to the airspace that can act as a proof of concept for future research.

Outcomes are summarized below.

(1) In order to better understand which types of mapf algorithms are best suited for use with UAVs, an in-depth overview and analysis of multi-agent pathfinding and some state of the art algorithms was provided. This showed that many types of algorithms, such as rule-based and SAT algorithms are either too slow, incomplete, or only work with certain types of graphs and should therefore not be considered. It also showed that multi-level mapf algorithms — algorithms with high-level and low-level searches such as ICTS — are likely to perform well at the scale at which UAVs are projected to operate. Finally, this analysis showed that frameworks such as Independence Detection, which attempt to reduce the number of agents whose paths are simultaneously found, are very beneficial due to the fact that agents will not typically be concentrated in a small area and therefore paths can be found individually.

(2) A*+OD+ID and CBS along with its suboptimal variants were implemented and experimentally tested. A*+OD+ID proved inadequate to handle the large number of simultaneous agents and can thus be safely ruled out when deciding which mapf algorithms to use. Unexpectedly, the greedy version of CBS was the slowest, although all three of CBS, GCBS and ECBS produced roughly the same results in a similar amount of time, all of which were adequate. This result also showed that optimal algorithms may very well be fast enough for use with UAVs, due to the open maps in which they operate. This is an ideal scenario as shorter paths are financially beneficial to anyone operating UAVs.

(3) A custom adaptation of online mapf applied to the rerouting of UAVs (UAVReroute) was designed, implemented and experimentally tested. Although conflict free paths for all agents were not found, the results were still promising and showed that online mapf along with the independence detection framework will likely prove to be a good strategy to use for contingency planning in the future. The test results also showed the necessity of a good dynamic rerouting algorithm, as conflicts still remained when single agent A* was used, which is an unacceptable result due to safety concerns of UAV operations.

6.3 Future Work

Due to time constraints at my internship I did not get to accomplish everything I would have liked. The following are the main items I did not have time to complete and are good areas of future work.

Further improve the dynamic online rerouting algorithm so that conflict-free paths for all active agents can be found in an acceptable amount of time. Specifically, Online Independence Detection looks promising and should continue to be tested with various other mapf algorithms that are designed to handle large obstacles, such as FA-A* [Cao et al., 2017]. I would also test other multi-level mapf algorithms, such as ICTS [Sharon et al., 2013] and Meta-Agent CBS [Sharon et al., 2012]. Although CBS seems like a good choice, other algorithms should also be tested to see how they perform when used in conjunction with the OID framework.

More robust testing of the dynamic inflight rerouting algorithm, especially stress testing, should also be performed. This should be done in order to see how the algorithm performs under extreme conditions.

One aspect of pathfinding for UAVs not yet mentioned in this thesis is the consideration of UAV dynamics and kinematics. Holonomic UAVs can hover and change direction instantaneously. Non-holonomic UAVs have fixed wings (like airplanes) and as such have much more restrictive equations of motion. It seems likely that the majority of UAVs will be holonomic, but for the sake of completeness, non-holonomic UAVs should also be considered in future work.

I currently have no plans to publish this work, but have not ruled it out and may do so in the future when time permits.

References

Antal, C., Granichin, O., and Levi, S. (2010). Adaptive autonomous soaring of multiple uavs using simultaneous perturbation stochastic approximation. In *49th IEEE Conference on Decision and Control (CDC)*, pages 3656–3661. IEEE.

Bareiss, D. and van den Berg, J. (2015). Generalized reciprocal collision avoidance. *The International Journal of Robotics Research*, 34(12):1501–1514.

Barer, M., Sharon, G., Stern, R., and Felner, A. (2014). Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem. In *Seventh Annual Symposium on Combinatorial Search*.

Boyarski, E., Felner, A., Stern, R., Sharon, G., Tolpin, D., Betzalel, O., and Shimony, E. (2015). Icbs: improved conflict-based search algorithm for multi-agent pathfinding. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*.

Brown, R. G., Hwang, P. Y., et al. (1992). *Introduction to random signals and applied Kalman filtering*, volume 3. Wiley New York.

Cao, P., Fan, Z., Gao, R. X., and Tang, J. (2017). A focal any-angle path-finding algorithm based on a* on visibility graphs. *arXiv preprint arXiv:1706.03144*.

Chouhan, S. S. and Niyogi, R. (2015). Dmapp: A distributed multi-agent path planning algorithm. In *Australasian Joint Conference on Artificial Intelligence*, pages 123–135. Springer.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to algorithms (3rd ed.)*. MIT press.

Daniel, K., Nash, A., Koenig, S., and Felner, A. (2010). Theta*: Any-angle path planning on grids. *Journal of Artificial Intelligence Research*, 39:533–579.

Ericson, C. (2004). *Real-time collision detection*. CRC Press.

Felner, A., Goldenberg, M., Sharon, G., Stern, R., Beja, T., Sturtevant, N., Schaeffer, J., and Holte, R. (2012). Partial-expansion a* with selective node generation. In *Twenty-Sixth AAAI Conference on Artificial Intelligence*.

Felner, A., Stern, R., Shimony, S. E., Boyarski, E., Goldenberg, M., Sharon, G., Sturtevant, N., Wagner, G., and Surynek, P. (2017). Search-based optimal solvers for the multi-agent pathfinding problem: Summary and challenges. In *Tenth Annual Symposium on Combinatorial Search*.

Goldenberg, M., Felner, A., Stern, R., Sharon, G., Sturtevant, N., Holte, R. C., and Schaeffer, J. (2014). Enhanced partial expansion a. *Journal of Artificial Intelligence Research*, 50:141–187.

Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107.

Ho, F., Salta, A., Geraldes, R., Goncalves, A., Cavazza, M., and Prendinger, H. (2019). Multi-agent path finding for uav traffic management. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, pages 131–139. International Foundation for Autonomous Agents and Multiagent Systems.

Khorshid, M. M., Holte, R. C., and Sturtevant, N. R. (2011). A polynomial-time algorithm for non-optimal multi-agent pathfinding. In *Fourth Annual Symposium on Combinatorial Search*.

Kopardekar, P., Rios, J., Prevot, T., Johnson, M., Jung, J., and Robinson, J. E. (2016). Unmanned aircraft system traffic management (utm) concept of operations.

Korf, R. E. (1990). Real-time heuristic search. *Artificial intelligence*, 42(2-3):189–211.

Ma, H., Li, J., Kumar, T., and Koenig, S. (2017). Lifelong multi-agent path finding for online pickup and delivery tasks. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, pages 837–845. International Foundation for Autonomous Agents and Multiagent Systems.

Ma, H., Wagner, G., Felner, A., Li, J., Kumar, T., and Koenig, S. (2018). Multi-agent path finding with deadlines. *arXiv preprint arXiv:1806.04216*.

Majeed, A. and Lee, S. (2018). A fast global flight path planning algorithm based on space circumscription and sparse visibility graph for unmanned aerial vehicle. *Electronics*, 7(12):375.

Majumdar, A. and Tedrake, R. (2013). Robust online motion planning with regions of finite time invariance. In *Algorithmic Foundations of Robotics X*, pages 543–558. Springer.

Mandalika, A., Salzman, O., and Srinivasa, S. (2018). Lazy receding horizon a* for efficient path planning in graphs with expensive-to-evaluate edges. In *Twenty-Eighth International Conference on Automated Planning and Scheduling*.

Ryan, M. (2010). Constraint-based multi-robot path planning. In *2010 IEEE International Conference on Robotics and Automation*, pages 922–928. IEEE.

Sajid, Q., Luna, R., and Bekris, K. E. (2012). Multi-agent pathfinding with simultaneous execution of single-agent primitives. In *SoCS*.

Sharon, G., Stern, R., Felner, A., and Sturtevant, N. R. (2012). Meta-agent conflict-based search for optimal multi-agent path finding. *SoCS*, 1:39–40.

Sharon, G., Stern, R., Felner, A., and Sturtevant, N. R. (2015). Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, 219:40–66.

Sharon, G., Stern, R., Goldenberg, M., and Felner, A. (2013). The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence*, 195:470–495.

Silver, D. (2005). Cooperative pathfinding. *AIIDE*, 1:117–122.

Standley, T. S. (2010). Finding optimal solutions to cooperative pathfinding problems. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*.

Steve Bradford, U. T. (2018). Nasa utm concept of operations. https://utm.arc.nasa.gov/docs/2018-UTM-ConOps-v1.0.pdf.

Sturtevant, N. R. (2012). Benchmarks for grid-based pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):144–148.

Sunday, D. (2012). Geometry algorithms. http://geomalgorithms.com/index.html. [Online; accessed 31-August-2019].

Surynek, P. (2009). A novel approach to path planning for multiple robots in bi-connected graphs. In *2009 IEEE International Conference on Robotics and Automation*, pages 3613–3619. IEEE.

Surynek, P. (2010). An optimization variant of multi-robot path planning is intractable. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*.

Surynek, P., Felner, A., Stern, R., and Boyarski, E. (2016). Efficient sat approach to multi-agent path finding under the sum of costs objective. In *Proceedings of the Twenty-second European Conference on Artificial Intelligence*, pages 810–818. IOS Press.

Švancara, J., Vlk, M., Stern, R., Atzmon, D., and Barták, R. (2019). Online multi-agent pathfinding. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 7732–7739.

Wagner, G. and Choset, H. (2015). Subdimensional expansion for multirobot path planning. *Artificial Intelligence*, 219:1–24.

Walker, T. T., Chan, D. M., and Sturtevant, N. R. (2017). Using hierarchical constraints to avoid conflicts in multi-agent pathfinding. In *Twenty-Seventh International Conference on Automated Planning and Scheduling*.

Walker, T. T., Sturtevant, N. R., and Felner, A. (2018). Extended increasing cost tree search for non-unit cost domains. In *IJCAI*, pages 534–540.

Yan, F., Liu, Y.-S., and Xiao, J.-Z. (2013). Path planning in complex 3d environments using a probabilistic roadmap method. *International Journal of Automation and computing*, 10(6):525–533.

Yu, J. and LaValle, S. M. (2013). Planning optimal paths for multiple robots on graphs. In *2013 IEEE International Conference on Robotics and Automation*, pages 3612–3617. IEEE.